Intel® Fortran Compiler User and Reference Guides

Document Number: 304970-006US

Contents

Legal Information.....77
Getting Help and Support.....79

Chapter 1: Introduction
Introducing the Intel(R) Fortran Compiler.....81
Notational Conventions.....81
Related Information.....87

Part I: Building Applications

Chapter 2: Overview: Building Applications
.....93

Chapter 3: Introduction: Basic Concepts
Choosing Your Development Environment.....95
Invoking the Intel® Fortran Compiler.....95
Default Tools.....97
Specifying Alternative Tools and Locations.....99
Compilation Phases.....99
Compiling and Linking for Optimization.....100
Compiling and Linking Multithread Programs.....101
What the Compiler Does by Default.....102
Generating Listing and Map Files.....103
Saving Compiler Information in your Executable.....104

Chapter 4: Building Applications from the Command Line
Using the Compiler and Linker from the Command Line.....107
Syntax for the ifort Command.....108
Examples of the ifort Command.....109
Creating, Running, and Debugging an Executable Program.....110
Redirecting Command-Line Output to Files.....113
Using Makefiles to Compile Your Application.....114
Specifying Memory Models to use with Systems Based on Intel® 64 Architecture.....115
Allocating Common Blocks.....116
Running Fortran Applications from the Command Line.....118

Chapter 5: Input and Output Files
Understanding Input File Extensions.....121
Producing Output Files.....122
Temporary Files Created by the Compiler or Linker.....124

Chapter 6: Setting Environment Variables
Using the ifortvars File to Specify Location of Components.....127
Setting Compile-Time Environment Variables.....129
Setting Run-Time Environment Variables.....132

Chapter 7: Using Compiler Options
Compiler Options Overview.....137
Using the Option Mapping Tool.....138
Compiler Directives Related to Options.....140

Chapter 8: Preprocessing
Using the fpp Preprocessor.....143
Using fpp Directives.....146
Using Predefined Preprocessor Symbols.....153

Chapter 9: Using Configuration Files and Response Files

Configuration Files and Response Files Overview.....157
Using Configuration Files.....157
Using Response Files.....158

Chapter 10: Debugging

Debugging Fortran Programs.....161
Preparing Your Program for Debugging.....161
Locating Unaligned Data.....164
Debugging a Program that Encounters a Signal or Exception...164
Debugging and Optimizations.....165
Debugging Multithreaded Programs.....168

Chapter 11: Data and I/O

Data Representation.....171
Data Representation Overview.....171
Integer Data Representations.....174
Logical Data Representations.....176
Character Representation.....178
Hollerith Representation.....179
Using Traceback Information.....181
Supported Native and Nonnative Numeric Formats.....181
Porting Nonnative Data.....187
Specifying the Data Format.....188
Fortran I/O.....197
Devices and Files Overview.....197
Logical Devices.....197
Types of I/O Statements.....201
Forms of I/O Statements.....203
Assigning Files to Logical Units.....205
File Organization.....208
Internal Files and Scratch Files.....209
File Access and File Structure.....210
File Records.....213

Record Types.....213
Record Length.....222
Record Access.....223
Record Transfer.....225
Specifying Default Pathnames and File Names.....227
Opening Files: OPEN Statement.....228
Obtaining File Information: INQUIRE Statement.....232
Closing Files: CLOSE Statement.....234
Record I/O Statement Specifiers.....235
File Sharing on Linux* OS and Mac OS* X Systems.....236
Specifying the Initial Record Position.....237
Advancing and Nonadvancing Record I/O.....238
User-Supplied OPEN Procedures: USEROPEN Specifier...238
Microsoft Fortran PowerStation Compatible Files.....246
Using Asynchronous I/O.....253

Chapter 12: Structuring Your Program

Structuring Your Program Overview.....257
Creating Fortran Executables.....257
Using Module (.mod) Files.....258
Using Include Files.....260
Advantages of Internal Procedures.....261
Storing Object Code in Static Libraries.....261
Storing Routines in Shareable Libraries.....261

Chapter 13: Programming with Mixed Languages

Programming with Mixed Languages Overview.....263
Calling Subprograms from the Main Program.....263
Summary of Mixed-Language Issues.....264
Adjusting Calling Conventions in Mixed-Language Programming.....266
Adjusting Calling Conventions in Mixed-Language Programming Overview.....266

ATTRIBUTES Properties and Calling Conventions.....268
Adjusting Naming Conventions in Mixed-Language Programming.....274
Adjusting Naming Conventions in Mixed-Language Programming Overview.....274
C/C++ Naming Conventions.....274
Procedure Names for Fortran, C, C++, and MASM.....275
Reconciling the Case of Names.....276
Fortran Module Names and ATTRIBUTES.....278
Prototyping a Procedure in Fortran.....279
Exchanging and Accessing Data in Mixed-Language Programming.....280
Exchanging and Accessing Data in Mixed-Language Programming.....280
Passing Arguments in Mixed-Language Programming.....281
Using Modules in Mixed-Language Programming.....283
Using Common External Data in Mixed-Language Programming.....285
Handling Data Types in Mixed-Language Programming.....289
Handling Data Types in Mixed-Language Programming Overview.....289
Handling Numeric, Complex, and Logical Data Types.....290
Handling Fortran Array Pointers and Allocatable Arrays.....293
Handling Integer Pointers.....294
Handling Arrays and Fortran Array Descriptors.....295
Handling Character Strings.....301
Handling User-Defined Types.....305
Intel(R) Fortran/Visual Basic* Mixed-Language Programs.....306
Interoperability with C.....306
Compiling and Linking Intel® Fortran/C Programs.....311
Calling C Procedures from an Intel® Fortran Program.....312

Chapter 14: Using Libraries

Supplied Libraries.....315
Creating Static Libraries.....319
Creating Shared Libraries.....321
Calling Library Routines.....323
Portability Considerations.....325
Portability Library Overview.....325
Using the IFPORT Portability Module.....325
Portability Routines.....326
Math Libraries.....329

Chapter 15: Error Handling

Handling Compile Time Errors.....331
Understanding Errors During the Build Process.....331
Compiler Message Catalog Support.....335
Using Source Code Verification.....336
Handling Run-Time Errors.....351
Understanding Run-Time Errors.....351
Run-Time Default Error Processing.....353
Run-Time Message Display and Format.....353
Values Returned at Program Termination.....356
Methods of Handling Errors.....357
Using the END, EOR, and ERR Branch Specifiers.....357
Using the IOSTAT Specifier and Fortran Exit Codes.....358
Locating Run-Time Errors.....359
List of Run-Time Error Messages.....361
Signal Handling (Linux* OS and Mac OS* X only).....410
Overriding the Default Run-Time Library Exception Handler.....411
Using Traceback Information.....412

Chapter 16: Portability Considerations

Portability Considerations Overview.....429
Understanding Fortran Language Standards.....429
Understanding Fortran Language Standards Overview...429
Using Standard Features and Extensions.....430
Using Compiler Optimizations.....431
Minimizing Operating System-Specific Information.....432
Storing and Representing Data.....432
Formatting Data for Transportability.....433

Chapter 17: Troubleshooting
Troubleshooting Your Application.....435

Chapter 18: Reference Information
Key Compiler Files Summary.....437
Compiler Limits.....438

Part II: Compiler Options

Chapter 19: Overview: Compiler Options
New Options.....444
Deprecated and Removed Compiler Options.....457

Chapter 20: Alphabetical Compiler Options
Compiler Option Descriptions and General Rules.....465
0 - 9
1.....469
4I2, 4I4, 4I8.....469
4L72, 4L80, 4L132.....469
4Na, 4Ya.....469
4Naltparam, 4Yaltparam.....469
4Nb,4Yb.....469
4Nd,4Yd.....469
4Nf.....469

4Nportlib, 4Yportlib.....469
4Ns,4Ys.....471
4R8,4R16.....471
4Yf.....471
4Nportlib, 4Yportlib.....471
66.....472
72,80,132.....472

A

align.....472
allow.....476
altparam.....478
ansi-alias, Qansi-alias.....479
arch.....480
architecture.....483
asmattr.....483
asmfile.....485
assume.....486
auto, Qauto.....496
auto-scalar, Qauto-scalar.....496
autodouble, Qautodouble.....498
automatic.....498
ax, Qax.....500

B

B.....503
Bdynamic.....504
bigobj.....506
bintext.....507
Bstatic.....508

C

c.....509
C.....510
CB.....510

ccdefault....510
check....511
cm....516
common-args, Qcommon-args....516
compile-only....516
complex-limited-range, Qcomplex-limited-range....516
convert....517
cpp, Qcpp....520
CU....520
cxxlib....520

D

D....522
d-lines, Qd-lines....523
dbglibs....524
DD....526
debug (Linux* OS and Mac OS* X)....526
debug (Windows* OS)....529
debug-parameters....532
define....533
diag, Qdiag....533
diag-dump, Qdiag-dump....538
diag, Qdiag....539
diag-enable sc-include, Qdiag-enable:sc-include....544
diag-enable sc-parallel, Qdiag-enable:sc-parallel....545
diag-error-limit, Qdiag-error-limit....547
diag-file, Qdiag-file....548
diag-file-append, Qdiag-file-append....550
diag-id-numbers, Qdiag-id-numbers....551
diag-once, Qdiag-once....552
dll....553
double-size....554
dps, Qdps....556

dryrun....556
dumpmachine....557
dynamic-linker....558
dynamiclib....559
dyncom, Qdyncom....560

E

E....561
e90, e95, e03....562
EP....562
error-limit....563
exe....563
extend-source....565
extfor....566
extfpp....567
extlnk....568

F

F (Windows*)....569
f66....570
f77rtl....572
Fa....573
FA....573
falias....573
falign-functions, Qfnalign....574
falign-stack....575
fast....577
fast-transcendentals, Qfast-transcendentals....578
fcode-asm....580
Fe....581
fexceptions....581
ffnalias....582
FI....583
finline....583

finline-functions....584
finline-limit....585
finstrument-functions, Qinstrument-functions....586
fixed....588
fkeep-static-consts, Qkeep-static-consts....589
fltconsistency....590
Fm....593
fma, Qfma....593
fmath-errno....594
fminshared....596
fnsplit, Qfnsplit....597
fomit-frame-pointer, Oy....598
Fo....600
fomit-frame-pointer, Oy....600
fp-model, fp....601
fp-model, fp....606
fp-port, Qfp-port....611
fp-relaxed, Qfp-relaxed....612
fp-speculation, Qfp-speculation....613
fp-stack-check, Qfp-stack-check 615
fpconstant....616
fpe....617
fpe-all....620
fpic....623
fpie....624
fpp, Qfpp....625
fpscomp....627
FR....637
fr32....637
free....638
fsource-asm....639
fstack-security-check, GS....640

fstack-security-check, GS....641
fsyntax-only....642
ftrapuv, Qtrapuv....642
ftz, Qftz....643
func-groups....646
funroll-loops....646
fverbose-asm....646
fvisibility....647

G

g, Zi, Z7....650
G2, G2-p9000....651
G5, G6, G7....653
gdwarf-2....655
Ge....656
gen-interfaces....657
global-hoist, Qglobal-hoist....658
Gm....660
Gs....660
fstack-security-check, GS....661
Gz....662

H

heap-arrays....662
help....663
homeparams....665
hotpatch....666

I

I....667
i-dynamic....669
i-static....669
i2, i4, i8....669
idirafter....669
iface....670

implicitnone....674
include....674
inline....674
inline-debug-info, Qinline-debug-info....676
inline-factor, Qinline-factor....677
inline-forceinline, Qinline-forceinline....679
inline-level, Ob....680
inline-max-per-compile, Qinline-max-per-compile....682
inline-max-per-routine, Qinline-max-per-routine....683
inline-max-size, Qinline-max-size....685
inline-max-total-size, Qinline-max-total-size....687
inline-min-size, Qinline-min-size....688
intconstant....690
integer-size....691
ip, Qip....693
ip-no-inlining, Qip-no-inlining....694
ip-no-pinlining, Qip-no-pinlining....695
IPF-flt-eval-method0, QIPF-flt-eval-method0....696
IPF-fltacc, QIPF-fltacc....698
IPF-fma, QIPF-fma....699
IPF-fp-relaxed, QIPF-fp-relaxed....699
ipo, Qipo....699
ipo-c, Qipo-c....701
ipo-jobs, Qipo-jobs....702
ipo-S, Qipo-S....704
ipo-separate, Qipo-separate....705
isystem....706
ivdep-parallel, Qivdep-parallel....707

L

l....708
L....709
LD....710

libdir....710
libs....712
link....715
logo....716
lowercase, Qlowercase....717

M

m....717
m32, m64....719
map....720
map-opts, Qmap-opts....721
march....723
mcmodel....724
mcpu....726
MD....726
MDs....728
mdynamic-no-pic....729
MG....730
mieee-fp....730
minstruction, Qinstruction....730
mixed-str-len-arg....732
mkl, Qmkl....732
ML....733
module....734
mp....735
multiple-processes, MP....735
mp1, Qprec....737
mrelax....738
MT....739
mtune....740
multiple-processes, MP....743
MW....744
MWs....744

N

names....744
nbs....746
no-bss-init, Qnobss-init....746
nodefaultlibs....747
nodefine....748
nofor-main....748
noinclude....749
nolib-inline....749
nostartfiles....750
nostdinc....751
nostdlib....751
nus....752

O

o....752
O....753
inline-level, Ob....758
object....760
Od....761
Og....763
onetrip, Qonetrip....764
Op....765
openmp, Qopenmp....765
openmp-lib, Qopenmp-lib....766
openmp-link, Qopenmp-link....768
openmp-profile, Qopenmp-profile....770
openmp-report, Qopenmp-report....771
openmp-stubs, Qopenmp-stubs....772
openmp-threadprivate, Qopenmp-threadprivate....774
opt-block-factor, Qopt-block-factor....775
opt-jump-tables, Qopt-jump-tables....776
opt-loadpair, Qopt-loadpair....778

opt-malloc-options.....779
opt-mem-bandwidth, Qopt-mem-bandwidth.....780
opt-mod-versioning, Qopt-mod-versioning.....782
opt-multi-version-aggressive,
Qopt-multi-version-aggressive.....783
opt-prefetch, Qopt-prefetch.....784
opt-prefetch-initial-values,
Qopt-prefetch-initial-values.....786
opt-prefetch-issue-excl-hint,
Qopt-prefetch-issue-excl-hint.....787
opt-prefetch-next-iteration,
Qopt-prefetch-next-iteration.....788
opt-ra-region-strategy, Qopt-ra-region-strategy.....790
opt-report, Qopt-report.....791
opt-report-file, Qopt-report-file.....793
opt-report-help, Qopt-report-help.....794
opt-report-phase, Qopt-report-phase.....795
opt-report-routine, Qopt-report-routine.....796
opt-streaming-stores, Qopt-streaming-stores.....797
opt-subscript-in-range, Qopt-subscript-in-range.....799
optimize.....800
Os.....800
Ot.....802
Ox.....803
fomit-frame-pointer, Oy.....803

P

p.....805
P.....806
pad, Qpad.....806
pad-source, Qpad-source.....807
par-affinity, Qpar-affinity.....808
par-num-threads, Qpar-num-threads.....810

par-report, Qpar-report....811
par-runtime-control, Qpar-runtime-control....813
par-schedule, Qpar-schedule....814
par-threshold, Qpar-threshold....818
parallel, Qparallel....819
pc, Qpc....821
pdbfile....822
pg....823
pie....823
prec-div, Qprec-div....825
prec-sqrt, Qprec-sqrt....826
preprocess-only....827
print-multi-lib....828
prof-data-order, Qprof-data-order....829
prof-dir, Qprof-dir....830
prof-file, Qprof-file....832
prof-func-groups....833
prof-func-order, Qprof-func-order....834
prof-gen, Qprof-gen....836
prof-genx, Qprof-genx....838
prof-hotness-threshold, Qprof-hotness-threshold....838
prof-src-dir, Qprof-src-dir....840
prof-src-root, Qprof-src-root....841
prof-src-root-cwd, Qprof-src-root-cwd....843
prof-use, Qprof-use....845

Q

ansi-alias, Qansi-alias....847
auto, Qauto....848
auto-scalar, Qauto-scalar....848
autodouble, Qautodouble....850
ax, Qax....850
Qchkstk....853

common-args, Qcommon-args.....855
complex-limited-range, Qcomplex-limited-range.....855
cpp, Qcpp.....856
d-lines, Qd-lines.....856
diag, Qdiag.....857
diag-dump, Qdiag-dump.....862
diag, Qdiag.....863
diag-enable sc-include, Qdiag-enable:sc-include.....867
diag-enable sc-parallel, Qdiag-enable:sc-parallel.....869
diag-error-limit, Qdiag-error-limit.....871
diag-file, Qdiag-file.....872
diag-file-append, Qdiag-file-append.....873
diag-id-numbers, Qdiag-id-numbers.....875
diag-once, Qdiag-once.....876
dps, Qdps.....877
dyncom, Qdyncom.....877
Qextend-source.....879
fast-transcendentals, Qfast-transcendentals.....879
fma, Qfma.....880
falign-functions, Qfnalign.....882
fnsplit, Qfnsplit.....883
fp-port, Qfp-port.....884
fp-relaxed, Qfp-relaxed.....885
fp-speculation, Qfp-speculation.....886
fp-stack-check, Qfp-stack-check 888
fpp, Qfpp.....889
ftz, Qftz.....891
global-hoist, Qglobal-hoist.....893
QIA64-fr32.....894
QIfist.....895
Qimsl.....895
inline-debug-info, Qinline-debug-info.....896

Qinline-dllimport.....897
inline-factor, Qinline-factor.....898
inline-forceinline, Qinline-forceinline.....900
inline-max-per-compile, Qinline-max-per-compile.....901
inline-max-per-routine, Qinline-max-per-routine.....903
inline-max-size, Qinline-max-size.....905
inline-max-total-size, Qinline-max-total-size.....906
inline-min-size, Qinline-min-size.....908
Qinstall.....910
minstruction, Qinstruction.....911
finstrument-functions, Qinstrument-functions.....912
ip, Qip.....914
ip-no-inlining, Qip-no-inlining.....915
ip-no-pinlining, Qip-no-pinlining.....916
IPF-flt-eval-method0, QIPF-flt-eval-method0.....917
IPF-fltacc, QIPF-fltacc.....919
IPF-fma, QIPF-fma.....920
IPF-fp-relaxed, QIPF-fp-relaxed.....920
ipo, Qipo.....920
ipo-c, Qipo-c.....922
ipo-jobs, Qipo-jobs.....923
ipo-S, Qipo-S.....925
ipo-separate, Qipo-separate.....926
ivdep-parallel, Qivdep-parallel.....927
fkeep-static-consts, Qkeep-static-consts.....928
Qlocation.....929
lowercase, Qlowercase.....931
map-opts, Qmap-opts.....931
mkl, Qmkl.....933
no-bss-init, Qnobss-init.....934
onetrip, Qonetrip.....935
openmp, Qopenmp.....936

openmp-lib, Qopenmp-lib.....937
openmp-link, Qopenmp-link.....939
openmp-profile, Qopenmp-profile.....940
openmp-report, Qopenmp-report.....942
openmp-stubs, Qopenmp-stubs.....943
openmp-threadprivate, Qopenmp-threadprivate.....944
opt-block-factor, Qopt-block-factor.....946
opt-jump-tables, Qopt-jump-tables.....947
opt-loadpair, Qopt-loadpair.....948
opt-mem-bandwidth, Qopt-mem-bandwidth.....949
opt-mod-versioning, Qopt-mod-versioning.....951
opt-multi-version-aggressive,
Qopt-multi-version-aggressive.....952
opt-prefetch, Qopt-prefetch.....953
opt-prefetch-initial-values,
Qopt-prefetch-initial-values.....955
opt-prefetch-issue-excl-hint,
Qopt-prefetch-issue-excl-hint.....956
opt-prefetch-next-iteration,
Qopt-prefetch-next-iteration.....957
opt-ra-region-strategy, Qopt-ra-region-strategy.....959
opt-report, Qopt-report.....960
opt-report-file, Qopt-report-file.....962
opt-report-help, Qopt-report-help.....963
opt-report-phase, Qopt-report-phase.....964
opt-report-routine, Qopt-report-routine.....965
opt-streaming-stores, Qopt-streaming-stores.....966
opt-subscript-in-range, Qopt-subscript-in-range.....968
Qoption.....969
qp.....971
pad, Qpad.....971
pad-source, Qpad-source.....972

Qpar-adjust-stack.....974
par-affinity, Qpar-affinity.....975
par-num-threads, Qpar-num-threads.....977
par-report, Qpar-report.....978
par-runtime-control, Qpar-runtime-control.....979
par-schedule, Qpar-schedule.....980
par-threshold, Qpar-threshold.....984
parallel, Qparallel.....986
pc, Qpc.....987
mp1, Qprec.....989
prec-div, Qprec-div.....990
prec-sqrt, Qprec-sqrt.....991
prof-data-order, Qprof-data-order.....992
prof-dir, Qprof-dir.....994
prof-file, Qprof-file.....995
prof-func-order, Qprof-func-order.....996
prof-gen, Qprof-gen.....998
prof-genx, Qprof-genx.....1000
prof-hotness-threshold, Qprof-hotness-threshold.....1000
prof-src-dir, Qprof-src-dir.....1001
prof-src-root, Qprof-src-root.....1003
prof-src-root-cwd, Qprof-src-root-cwd.....1005
prof-use, Qprof-use.....1006
rcd, Qrcd.....1008
rct, Qrct.....1009
safe-cray-ptr, Qsafe-cray-ptr.....1010
save, Qsave.....1012
save-temps, Qsave-temps.....1013
scalar-rep, Qscalar-rep.....1015
Qsfalign.....1016
sox, Qsox.....1017
tcheck, Qtcheck.....1019

tcollect, Qtcollect....1020
tcollect-filter, Qtcollect-filter....1021
tprofile, Qtprofile....1023
ftrapuv, Qtrapuv....1024
unroll, Qunroll....1026
unroll-aggressive, Qunroll-aggressive....1027
uppercase, Quppercase....1028
use-asm, Quse-asm....1028
Quse-msasm-symbols....1029
Quse-vcdebug....1030
Qvc....1031
vec, Qvec....1032
vec-guard-write, Qvec-guard-write....1033
vec-report, Qvec-report....1034
vec-threshold, Qvec-threshold....1036
x, Qx....1038
zero, Qzero....1042

R

r8, r16....1044
rcd, Qrcd....1044
rct, Qrct....1045
real-size....1046
recursive....1047
reentrancy....1049
RTCu....1050

S

S....1050
safe-cray-ptr, Qsafe-cray-ptr....1051
save, Qsave....1053
save-temps, Qsave-temps....1054
scalar-rep, Qscalar-rep....1056
shared....1057

shared-intel....1058
shared-libgcc....1060
source....1061
sox, Qsox....1062
stand....1063
static....1065
staticlib....1066
static-intel....1068
static-libgcc....1069
std, std90, std95, std03....1070
std, std90, std95, std03....1070
std, std90, std95, std03....1070
std, std90, std95, std03....1070
syntax-only....1070

T

T....1071
tcheck, Qtcheck....1072
tcollect, Qtcollect....1073
tcollect-filter, Qtcollect-filter....1075
Tf....1077
threads....1077
tprofile, Qtprofile....1078
traceback....1080
tune....1081

U

u (Linux* and Mac OS* X)....1083
u (Windows*)....1083
U....1084
undefine....1085
unroll, Qunroll....1085
unroll-aggressive, Qunroll-aggressive....1086
uppercase, Quppercase....1088

us....1088
use-asm, Quse-asm....1088
V
v....1089
V (Linux* and Mac OS* X)....1090
V (Windows*)....1090
vec, Qvec....1090
vec-guard-write, Qvec-guard-write....1091
vec-report, Qvec-report....1092
vec-threshold, Qvec-threshold....1094
vms....1095
W
w....1098
W0, W1....1098
W0, W1....1098
Wa....1098
warn....1099
watch....1105
WB....1106
what....1107
winapp....1108
Winline....1109
Wl....1110
Wp....1111
X
x, Qx....1112
X....1116
Xlinker....1118
Y
y....1119
Z
g, Zi, Z7....1119

Zd....1121
zero, Qzero....1121
g, Zi, Z7....1122
Zl....1124
Zp....1124
Zs....1124
Zx....1124

Chapter 21: Quick Reference Guides and Cross References

Windows* OS Quick Reference Guide and Cross Reference...1127
Linux* OS and Mac OS* X Quick Reference Guide and Cross Reference....1178

Chapter 22: Related Options

Linking Tools and Options....1229
Fortran Preprocessor Options....1232

Part III: Optimizing Applications

Chapter 23: Intel(R) Fortran Optimizing Applications

Overview: Optimizing Applications....1239
Optimizing with the Intel® Compiler....1239
Optimizing for Performance....1241
Overview of Parallelism Method....1242
Quick Reference Lists....1244
Other Resources....1245

Chapter 24: Evaluating Performance

Performance Analysis....1247
Using a Performance Enhancement Methodology....1247
Intel® Performance Analysis Tools and Libraries....1250
Performance Enhancement Strategies....1251

Using Compiler Reports.....1258
Compiler Reports Overview.....1258
Compiler Reports Quick Reference.....1258
Generating Reports.....1260
Interprocedural Optimizations (IPO) Report.....1263
Profile-guided Optimization (PGO) Report.....1269
High-level Optimization (HLO) Report.....1273
High Performance Optimizer (HPO) Report.....1286
Parallelism Report.....1287
Software Pipelining (SWP) Report (Linux* and Windows*).....1288
Vectorization Report.....1294
OpenMP* Report.....1299

Chapter 25: Using Compiler Optimizations

Automatic Optimizations Overview.....1301
Enabling Automatic Optimizations.....1302
Targeting IA-32 and Intel(R) 64 Architecture Processors Automatically.....1306
Targeting Multiple IA-32 and Intel(R) 64 Architecture Processors for Run-time Performance.....1310
Targeting IA-64 Architecture Processors Automatically.....1313
Restricting Optimizations.....1314

Chapter 26: Using Parallelism: OpenMP* Support

OpenMP* Support Overview.....1317
OpenMP* Options Quick Reference.....1318
OpenMP* Source Compatibility and Interoperability with Other Compilers.....1321
Using OpenMP*.....1323
Parallel Processing Model.....1326
Verifying OpenMP* Using Parallel Lint.....1331
OpenMP* Clauses".....1342

Data Scope Attribute Clauses Overview.....1342
Specifying Schedule Type and Chunk Size.....1342
COPYIN Clause.....1344
DEFAULT Clause.....1345
PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses....1346
REDUCTION Clause.....1348
SHARED Clause.....1351
OpenMP* Directives.....1352
Programming with OpenMP*.....1352
Combined Parallel and Worksharing Constructs.....1359
Parallel Region Directives.....1360
Synchronization Constructs.....1364
THREADPRIVATEthreadprivate Directive.....1370
Worksharing Construct Directives.....1371
Tasking Directives.....1374
OpenMP* Advanced Issues.....1375
OpenMP* Examples.....1379
Libraries, Directives, Clauses, and Environmental Variables...1383
OpenMP* Environment Variables.....1383
OpenMP* Directives and Clauses Summary.....1392
OpenMP* Library Support.....1398

Chapter 27: Using Parallelism: Automatic Parallelization

Auto-parallelization Overview.....1447
Auto-Parallelization Options Quick Reference.....1450
Auto-parallelization: Enabling, Options, Directives, and Environment Variables.....1451
Programming with Auto-parallelization.....1453
Programming for Multithread Platform Consistency.....1454

Chapter 28: Using Parallelism: Automatic Vectorization

Automatic Vectorization Overview.....1459
Automatic Vectorization Options Quick Reference.....1459

Programming Guidelines for Vectorization.....1461
Vectorization and Loops.....1462
Loop Constructs.....1466
Absence of Loop-carried Memory Dependency with IVDEP Directive.....1474
Vectorization Examples.....1475

Chapter 29: Using Parallelism: Multi-Threaded Applications

Creating Multithread Applications Overview.....1479
Basic Concepts of Multithreading.....1479
Developing Multithread Applications.....1480
Writing a Multithread Program Overview.....1480
Modules for Multithread Programs.....1481
Starting and Stopping Threads.....1481
Thread Routine Format.....1484
Sharing Resources.....1488
Thread Local Storage.....1491
Synchronizing Threads.....1491
Handling Errors in Multithread Programs.....1492
Table of Multithread Routines.....1492
Working with Multiple Processes.....1495

Chapter 30: Using Interprocedural Optimization (IPO)

Interprocedural Optimization (IPO) Overview.....1497
Interprocedural Optimization (IPO) Quick Reference.....1500
Using IPO.....1501
IPO-Related Performance Issues.....1504
IPO for Large Programs.....1504
Understanding Code Layout and Multi-Object IPO.....1506
Creating a Library from IPO Objects.....1508
Requesting Compiler Reports with the xi* Tools.....1510
Inline Expansion of Functions.....1511

Inline Function Expansion.....1511
Compiler Directed Inline Expansion of User Functions...1512
Developer Directed Inline Expansion of User Functions.....1514

Chapter 31: Using Profile-Guided Optimization (PGO)

Profile-Guided Optimizations Overview.....1519
Profile-Guided Optimization (PGO) Quick Reference.....1520
Profile an Application.....1530
PGO Tools.....1532
PGO Tools Overview.....1532
code coverage Tool.....1532
test prioritization Tool.....1552
profmerge and proforder Tools.....1561
Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations.....1566
Comparison of Function Order Lists and IPO Code Layout.....1573
PGO API Support.....1573
API Support Overview.....1573
PGO Environment Variables.....1574
Dumping Profile Information.....1576
Interval Profile Dumping.....1578
Resetting the Dynamic Profile Counters.....1579
Dumping and Resetting Profile Information.....1579

Chapter 32: Using High-Level Optimization (HLO)

High-Level Optimizations (HLO) Overview.....1581
Loop Unrolling.....1583
Loop Independence.....1584
Prefetching with Options.....1589

Chapter 33: Optimization Support Features

Optimization Support Features Overview.....1591
Loop Support.....1591
Loop Unrolling Support.....1595
Vectorization Support.....1596
Prefetching Support.....1602
Software Pipelining Support (IA-64 Architecture).....1605
About Register Allocation.....1606

Chapter 34: Programming Guidelines

Coding Guidelines for Intel® Architectures.....1611
Setting Data Type and Alignment.....1613
Using Arrays Efficiently.....1623
Improving I/O Performance.....1630
Improving Run-time Efficiency.....1636
Using Fortran Intrinsics.....1638
Understanding Run-time Performance.....1638
Understanding Data Alignment.....1642
Timing Your Application.....1643
Applying Optimization Strategies.....1647
Optimizing the Compilation Process.....1658
Optimizing the Compilation Process Overview.....1658
Efficient Compilation.....1658
Stacks: Automatic Allocation and Checking.....1660
Little-endian-to-Big-endian Conversion (IA-32 Architecture).....1664
Symbol Visibility Attribute Options (Linux* and Mac OS* X).....1669
Data Alignment Options.....1671

Part IV: Floating-point Operations

Chapter 35: Overview: Floating-point Operations

Chapter 36: Floating-point Options Quick Reference

Chapter 37: Understanding Floating-point Operations

Programming Tradeoffs in Floating-point Applications.....1681
Floating-point Optimizations.....1682
Using the -fp-model (/fp) Option.....1684
Denormal Numbers.....1688
Floating-point Environment.....1688
Setting the FTZ and DAZ Flags.....1689
Checking the Floating-point Stack State.....1691

Chapter 38: Tuning Performance

Overview: Tuning Performance.....1693
Avoiding Exact Floating-point Comparison.....1693
Handling Floating-point Array Operations in a Loop Body.....1694
Reducing the Impact of Denormal Exceptions.....1694
Avoiding Mixed Data Type Arithmetic Expressions.....1696
Using Efficient Data Types.....1698

Chapter 39: Handling Floating-point Exceptions

Overview: Controlling Floating-point Exceptions.....1699
Handling Floating-point Exceptions.....1700
File fordef.for and Its Usage.....1703
Setting and Retrieving Floating-point Status and Control Words (IA-32).....1706
Overview: Setting and Retrieving Floating-point Status and Control Word.....1706
Understanding Floating-point Status Word.....1709
Floating-point Control Word Overview.....1710
Using Exception, Precision, and Rounding Parameters...1711
Handling Floating-point Exceptions with the -fpe or /fpe Compiler Option.....1714

Using the -fpe or /fpe Compiler Options.....1714
Understanding the Impact of Application Types.....1717

Chapter 40: Understanding IEEE Floating-point Operations

Overview: Understanding IEEE Floating-point Standard.....1721
Floating-point Formats.....1721
Limitations of Numeric Conversion.....1721
Special Values.....1722
Representing Floating-point Numbers.....1724
Floating-point Representation.....1724
Retrieving Parameters of Numeric Representations.....1725
ULPs, Relative Error, and Machine Epsilon.....1727
Native IEEE Floating-point Representation.....1728
Handling Exceptions and Errors.....1732
Loss of Precision Errors.....1732
Rounding Errors.....1733

Part V: Language Reference

Chapter 41: Overview: Language Reference

New Language Features.....1739

Chapter 42: Conformance, Compatibility, and Fortran 2003 Features

Language Standards Conformance.....1741
Language Compatibility.....1741
Fortran 2003 Features.....1741

Chapter 43: Program Structure, Characters, and Source Forms

Program Structure.....1745
Statements.....1746

Names.....1748
Keywords.....1750
Character Sets.....1750
Source Forms.....1752
Free Source Form.....1754
Fixed and Tab Source Forms.....1757
Source Code Useable for All Source Forms.....1761

Chapter 44: Data Types, Constants, and Variables

Intrinsic Data Types.....1763
Integer Data Types.....1765
Real Data Types.....1769
Complex Data Types.....1774
General Rules for Complex Constants.....1775
COMPLEX(4) Constants.....1775
COMPLEX(8) or DOUBLE COMPLEX Constants.....1776
COMPLEX(16) Constants.....1777
Logical Data Types.....1778
Logical Constants.....1779
Character Data Type.....1779
Character Constants.....1780
C Strings in Character Constants.....1781
Character Substrings.....1783
Derived Data Types.....1784
Derived-Type Definition.....1785
Default Initialization.....1785
Structure Components.....1786
Structure Constructors.....1790
Binary, Octal, Hexadecimal, and Hollerith Constants.....1792
Binary Constants.....1792
Octal Constants.....1793
Hexadecimal Constants.....1793

Hollerith Constants.....1794
Determining the Data Type of Nondecimal Constants...1795
Variables.....1798
Data Types of Scalar Variables.....1799
Arrays.....1800

Chapter 45: Expressions and Assignment Statements

Expressions.....1817
Numeric Expressions.....1818
Character Expressions.....1823
Relational Expressions.....1823
Logical Expressions.....1825
Defined Operations.....1827
Summary of Operator Precedence.....1827
Initialization and Specification Expressions.....1828
Assignment Statements.....1833
Intrinsic Assignments.....1833
Defined Assignments.....1839
Pointer Assignments.....1840
WHERE Statement and Construct Overview.....1843
FORALL Statement and Construct Overview.....1843

Chapter 46: Specification Statements

Type Declaration Statements.....1846
Declaration Statements for Noncharacter Types.....1847
Declaration Statements for Character Types.....1849
Declaration Statements for Derived Types.....1852
Declaration Statements for Arrays.....1853
ALLOCATABLE Attribute and Statement Overview.....1862
ASYNCHRONOUS Attribute and Statement Overview.....1862
AUTOMATIC and STATIC Attributes and Statements Overview.....1862
BIND Attribute and Statement Overview.....1862

COMMON Statement Overview.....1862
DATA Statement Overview.....1863
DIMENSION Attribute and Statement Overview.....1863
EQUIVALENCE Statement Overview.....1863
Making Arrays Equivalent.....1863
Making Substrings Equivalent.....1865
EQUIVALENCE and COMMON Interaction.....1870
EXTERNAL Attribute and Statement Overview.....1872
IMPLICIT Statement Overview.....1872
INTENT Attribute and Statement Overview.....1872
INTRINSIC Attribute and Statement Overview.....1872
NAMELIST Statement Overview.....1873
OPTIONAL Attribute and Statement Overview.....1873
PARAMETER Attribute and Statement Overview.....1873
POINTER Attribute and Statement Overview.....1873
PROTECTED Attribute and Statement Overview.....1873
PUBLIC and PRIVATE Attributes and Statements Overview....1873
SAVE Attribute and Statement Overview.....1873
TARGET Attribute and Statement Overview.....1874
VALUE Attribute and Statement Overview.....1874
VOLATILE Attribute and Statement Overview.....1874

Chapter 47: Dynamic Allocation

ALLOCATE Statement Overview.....1875
Allocation of Allocatable Arrays.....1876
Allocation of Pointer Targets.....1877
DEALLOCATE Statement Overview.....1878
Deallocation of Allocatable Arrays.....1878
Deallocation of Pointer Targets.....1880
NULLIFY Statement Overview.....1881

Chapter 48: Execution Control

Branch Statements.....1883

Unconditional GO TO Statement Overview.....1884
Computed GO TO Statement Overview.....1884
The ASSIGN and Assigned GO TO Statements Overview.....1884
Arithmetic IF Statement Overview.....1885
CALL Statement Overview.....1885
CASE Constructs Overview.....1885
CONTINUE Statement Overview.....1885
DO Constructs Overview.....1885
Forms for DO Constructs.....1886
Execution of DO Constructs.....1886
DO WHILE Statement Overview.....1894
CYCLE Statement Overview.....1894
EXIT Statement Overview.....1894
END Statement Overview.....1894
IF Construct and Statement Overview.....1894
IF Construct Overview.....1894
IF Statement Overview.....1894
PAUSE Statement Overview.....1894
RETURN Statement Overview.....1895
STOP Statement Overview.....1895

Chapter 49: Program Units and Procedures

Main Program.....1898
Modules and Module Procedures.....1898
Module References.....1899
USE Statement.....1899
Intrinsic Modules.....1900
ISO_C_BINDING.....1901
ISO_FORTRAN_ENV.....1904
IEEE Intrinsic Modules and Procedures.....1906
Block Data Program Units.....1914

Functions, Subroutines, and Statement Functions.....1914
General Rules for Function and Subroutine Subprograms.....1915
Functions.....1916
Subroutines.....1918
Statement Functions.....1918
External Procedures.....1918
Internal Procedures.....1918
Argument Association.....1920
Optional Arguments.....1923
Array Arguments.....1924
Pointer Arguments.....1925
Assumed-Length Character Arguments.....1926
Character Constant and Hollerith Arguments.....1927
Alternate Return Arguments.....1928
Dummy Procedure Arguments.....1929
References to Generic Procedures.....1930
References to Non-Fortran Procedures.....1935
Procedure Interfaces.....1935
Determining When Procedures Require Explicit Interfaces.....1937
Defining Explicit Interfaces.....1938
Defining Generic Names for Procedures.....1938
Defining Generic Operators.....1940
Defining Generic Assignment.....1942
CONTAINS Statement Overview.....1944
ENTRY Statement Overview.....1944
ENTRY Statements in Function Subprograms.....1944
ENTRY Statements in Subroutine Subprograms.....1945
IMPORT Statement Overview.....1946

Chapter 50: Intrinsic Procedures

Argument Keywords in Intrinsic Procedures.....1949
Overview of Bit Functions.....1951
Categories and Lists of Intrinsic Procedures.....1953
Categories of Intrinsic Functions.....1953
Intrinsic Subroutines.....1975

Chapter 51: Data Transfer I/O Statements

Records and Files.....1979
Components of Data Transfer Statements.....1980
I/O Control List.....1981
I/O Lists.....1990
READ Statements.....1997
Forms for Sequential READ Statements.....1998
Forms for Direct-Access READ Statements.....2012
Forms for Stream READ Statements.....2014
Forms and Rules for Internal READ Statements.....2014
ACCEPT Statement Overview.....2017
WRITE Statements.....2017
Forms for Sequential WRITE Statements.....2017
Forms for Direct-Access WRITE Statements.....2026
Forms for Stream WRITE Statements.....2027
Forms and Rules for Internal WRITE Statements.....2027
PRINT and TYPE Statements Overview.....2028
REWRITE Statement Overview.....2029

Chapter 52: I/O Formatting

Format Specifications.....2031
Data Edit Descriptors.....2039
Forms for Data Edit Descriptors.....2040
General Rules for Numeric Editing.....2042
Integer Editing.....2044
Real and Complex Editing.....2049
Logical Editing (L).....2061

Character Editing (A).....2062
Default Widths for Data Edit Descriptors.....2065
Terminating Short Fields of Input Data.....2066
Control Edit Descriptors.....2068
Forms for Control Edit Descriptors.....2068
Positional Editing.....2070
Sign Editing.....2071
Blank Editing.....2073
Scale-Factor Editing (P).....2074
Slash Editing (/).....2077
Colon Editing (:).....2079
Dollar-Sign ($) and Backslash (\ ) Editing.....2079
Character Count Editing (Q).....2080
Character String Edit Descriptors.....2083
Character Constant Editing.....2083
H Editing.....2084
Nested and Group Repeat Specifications.....2086
Variable Format Expressions.....2086
Printing of Formatted Records.....2089
Interaction Between Format Specifications and I/O Lists.....2090

Chapter 53: File Operation I/O Statements

BACKSPACE Statement Overview.....2103
CLOSE Statement Overview.....2103
DELETE Statement Overview.....2103
ENDFILE Statement Overview.....2103
FLUSH Statement Overview.....2103
INQUIRE Statement Overview.....2103
ACCESS Specifier.....2104
ACTION Specifier.....2104
ASYNCHRONOUS Specifier.....2105
BINARY Specifier (W*32, W*64).....2105

BLANK Specifier.....2106
BLOCKSIZE Specifier.....2106
BUFFERED Specifier.....2106
CARRIAGECONTROL Specifier.....2107
CONVERT Specifier.....2107
DELIM Specifier.....2108
DIRECT Specifier.....2109
EXIST Specifier.....2109
FORM Specifier.....2110
FORMATTED Specifier.....2110
ID Specifier.....2111
IOFOCUS Specifier (W*32, W*64).....2111
MODE Specifier.....2112
NAME Specifier.....2112
NAMED Specifier.....2112
NEXTREC Specifier.....2113
NUMBER Specifier.....2113
OPENED Specifier.....2113
ORGANIZATION Specifier.....2114
PAD Specifier.....2114
PENDING Specifier.....2114
POS Specifier.....2115
POSITION Specifier.....2116
READ Specifier.....2116
READWRITE Specifier.....2117
RECL Specifier.....2117
RECORDTYPE Specifier.....2117
SEQUENTIAL Specifier.....2118
SHARE Specifier.....2119
UNFORMATTED Specifier.....2120
WRITE Specifier.....2120
OPEN Statement Overview.....2120

ACCESS Specifier.....2125
ACTION Specifier.....2125
ASSOCIATEVARIABLE Specifier.....2126
ASYNCHRONOUS Specifier.....2126
BLANK Specifier.....2127
BLOCKSIZE Specifier.....2127
BUFFERCOUNT Specifier.....2128
BUFFERED Specifier.....2128
CARRIAGECONTROL Specifier.....2129
CONVERT Specifier.....2130
DEFAULTFILE Specifier.....2133
DELIM Specifier.....2133
DISPOSE Specifier.....2134
FILE Specifier.....2134
FORM Specifier.....2136
IOFOCUS Specifier (W*32, W*64).....2137
MAXREC Specifier.....2138
MODE Specifier.....2138
NAME Specifier.....2138
NOSHARED Specifier.....2138
ORGANIZATION Specifier.....2138
PAD Specifier.....2139
POSITION Specifier.....2139
READONLY Specifier.....2140
RECL Specifier.....2141
RECORDSIZE Specifier.....2143
RECORDTYPE Specifier.....2143
SHARE Specifier.....2144
SHARED Specifier.....2146
STATUS Specifier.....2146
TITLE Specifier (W*32, W*64).....2147
TYPE Specifier.....2148

USEROPEN Specifier.....2148
REWIND Statement Overview.....2155
WAIT Statement Overview.....2155

Chapter 54: Compilation Control Lines and Statements

Chapter 55: Directive Enhanced Compilation
Syntax Rules for Compiler Directives.....2159
General Compiler Directives.....2160
Rules for General Directives that Affect DO Loops.....2162
Rules for Loop Directives that Affect Array Assignment Statements.....2163
OpenMP* Fortran Compiler Directives.....2164
Data Scope Attribute Clauses.....2166
Conditional Compilation Rules.....2167
Nesting and Binding Rules.....2168

Chapter 56: Scope and Association
Scope.....2171
Unambiguous Generic Procedure References.....2175
Resolving Procedure References.....2175
References to Generic Names.....2176
References to Specific Names.....2179
References to Nonestablished Names.....2180
Association.....2181
Name Association.....2183
Pointer Association.....2185
Storage Association.....2186

Chapter 57: Deleted and Obsolescent Language Features
Deleted Language Features in Fortran 95.....2191
Obsolescent Language Features in Fortran 95.....2191
Obsolescent Language Features in Fortran 90.....2193

Chapter 58: Additional Language Features

FORTRAN 66 Interpretation of the EXTERNAL Statement.....2195
Alternative Syntax for the PARAMETER Statement.....2198
Alternative Syntax for Binary, Octal, and Hexadecimal Constants.....2199
Alternative Syntax for a Record Specifier.....2200
Alternative Syntax for the DELETE Statement.....2200
Alternative Form for Namelist External Records.....2200
Record Structures.....2201
Structure Declarations.....2203
References to Record Fields.....2204
Aggregate Assignment.....2209

Chapter 59: Additional Character Sets

Character and Key Code Charts for Windows* OS.....2211
ASCII Character Codes for Windows* Systems.....2211
ANSI Character Codes for Windows* Systems.....2214
Key Codes for Windows* Systems.....2216
ASCII Character Set for Linux* OS and Mac OS* X.....2219

Chapter 60: Data Representation Models

Model for Integer Data.....2224
Model for Real Data.....2225
Model for Bit Data.....2227

Chapter 61: Run-Time Library Routines

Module Routines.....2229
OpenMP* Fortran Routines.....2230

Chapter 62: Summary of Language Extensions

Source Forms.....2233
Names.....2234

Character Sets.....2234
Intrinsic Data Types.....2234
Constants.....2235
Expressions and Assignment.....2235
Specification Statements.....2235
Execution Control.....2235
Program Units and Procedures.....2236
Compilation Control Lines and Statements.....2236
Built-In Functions.....2236
I/O Statements.....2236
I/O Formatting.....2237
File Operation Statements.....2237
Compiler Directives.....2239
Intrinsic Procedures.....2240
Additional Language Features.....2244
Run-Time Library Routines.....2245

Chapter 63: A to Z Reference

Language Summary Tables.....2248
Statements for Program Unit Calls and Definitions.....2249
Statements Affecting Variables.....2250
Statements for Input and Output.....2252
Compiler Directives.....2253
Program Control Statements and Procedures.....2258
Inquiry Intrinsic Functions.....2260
Random Number Intrinsic Procedures.....2262
Date and Time Intrinsic Subroutines.....2263
Keyboard and Speaker Library Routines.....2264
Statements and Intrinsic Procedures for Memory Allocation and Deallocation.....2264
Intrinsic Functions for Arrays.....2265
Intrinsic Functions for Numeric and Type Conversion...2267

Trigonometric, Exponential, Root, and Logarithmic Intrinsic Procedures....2269
Intrinsic Functions for Floating-Point Inquiry and Control....2273
Character Intrinsic Functions....2275
Intrinsic Procedures for Bit Operation and Representation....2277
QuickWin Library Routines (W*32, W*64)....2279
Graphics Library Routines (W*32, W*64)....2282
Portability Library Routines....2288
National Language Support Library Routines (W*32, W*64)....2301
POSIX* Library Procedures....2304
Dialog Library Routines (W*32, W*64)....2312
COM and Automation Library Routines (W*32, W*64))....2314
Miscellaneous Run-Time Library Routines....2317
Intrinsic Functions Not Allowed as Actual Arguments...2319
A to B....2319
ABORT....2319
ABOUTBOXQQ (W*32, W*64)....2320
ABS....2321
ACCEPT....2323
ACCESS....2324
ACHAR....2325
ACOS....2326
ACOSD....2327
ACOSH....2328
ADJUSTL....2328
ADJUSTR....2329
AIMAG....2330
AINT....2331

ALARM....2333
ALIAS....2334
ALL....2335
ALLOCATABLE....2337
ALLOCATE....2338
ALLOCATED....2340
AND....2342
ANINT....2342
ANY....2344
APPENDMENUQQ (W*32, W*64)....2345
ARC, ARC_W (W*32, W*64)....2348
ASIN....2350
ASIND....2351
ASINH....2352
ASSIGN - Label Assignment....2352
Assignment(=) - Defined Assignment....2354
Assignment - Intrinsic....2357
ASSOCIATED....2360
ASSUME_ALIGNED....2362
ASYNCHRONOUS....2363
ATAN....2365
ATAN2....2365
ATAN2D....2367
ATAND....2368
ATANH....2368
ATOMIC....2369
ATTRIBUTES....2371
AUTOAddArg (W*32, W*64)....2395
AUTOAllocateInvokeArgs (W*32, W*64)....2397
AUTODeallocateInvokeArgs (W*32, W*64)....2397
AUTOGetExceptInfo (W*32, W*64)....2398
AUTOGetProperty (W*32, W*64)....2398

AUTOGetPropertyByID (W*32, W*64).....2400
AUTOGetPropertyInvokeArgs (W*32, W*64).....2400
AUTOInvoke (W*32, W*64).....2401
AUTOMATIC.....2402
AUTOSetProperty (W*32, W*64).....2405
AUTOSetPropertyByID (W*32, W*64).....2406
AUTOSetPropertyInvokeArgs (W*32, W*64).....2407
BACKSPACE.....2407
BADDRESS.....2409
BARRIER.....2409
BEEPQQ.....2410
BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN.....2411
BIC, BIS.....2412
BIND.....2414
BIT.....2416
BIT_SIZE.....2416
BLOCK DATA.....2417
BSEARCHQQ.....2420
BTEST.....2422
BYTE.....2424
C to D.....2424
C_ASSOCIATED.....2424
C_F_POINTER.....2425
C_F_PROCPOINTER.....2426
C_FUNLOC.....2427
C_LOC.....2427
CACHESIZE.....2429
CALL.....2429
CASE.....2433
CDFLOAT.....2440
CEILING.....2441
CHANGEDIRQQ.....2442

CHANGEDRIVEQQ.....2443
CHAR.....2444
CHARACTER.....2445
CHDIR.....2446
CHMOD.....2449
CLEARSCREEN (W*32, W*64).....2451
CLEARSTATUSFPQQ.....2452
CLICKMENUQQ (W*32, W*64).....2455
CLOCK.....2456
CLOCKX.....2457
CLOSE.....2457
CMPLX.....2459
COMAddObjectReference (W*32, W*64).....2460
COMCLSIDFromProgID (W*32, W*64).....2461
COMCLSIDFromString (W*32, W*64).....2461
COMCreateObjectByGUID (W*32, W*64).....2462
COMCreateObjectByProgID (W*32, W*64).....2463
COMGetActiveObjectByGUID (W*32, W*64).....2463
COMGetActiveObjectByProgID (W*32, W*64).....2464
COMGetFileObject (W*32, W*64).....2465
COMInitialize (W*32, W*64).....2465
COMIsEqualGUID (W*32, W*64).....2468
COMMAND_ARGUMENT_COUNT.....2468
COMMITQQ.....2471
COMMON.....2473
COMPLEX.....2478
COMPLINT, COMPLREAL, COMPLLOG.....2479
COMQueryInterface (W*32, W*64).....2479
COMReleaseObject (W*32, W*64).....2480
COMStringFromGUID (W*32, W*64).....2481
COMUninitialize (W*32, W*64).....2482
CONJG.....2482

CONTAINS....2483
CONTINUE....2484
COPYIN....2485
COPYPRIVATE....2485
COS....2486
COSD....2487
COSH....2488
COTAN....2488
COTAND....2489
COUNT....2490
CPU_TIME....2492
CRITICAL....2492
CSHIFT....2494
CSMG....2497
CTIME....2497
CYCLE....2498
DATA....2500
DATE Intrinsic Procedure....2505
DATE Portability Routine....2507
DATE4....2508
DATE_AND_TIME....2509
DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN....2511
DBLE....2513
DCLOCK....2515
DCMPLX....2516
DEALLOCATE....2517
DECLARE and NODECLARE....2518
DECODE....2519
DEFAULT....2521
DEFINE....2522
DEFINE FILE....2523

DELDIRQQ....2525
DELETE....2526
DELETEMENUQQ (W*32, W*64)....2527
DELFILESQQ....2528
Derived Type (TYPE)....2530
DFLOAT....2536
DFLOATI, DFLOATJ, DFLOATK....2537
DIGITS....2538
DIM....2539
DIMENSION....2540
DISPLAYCURSOR....2543
DISTRIBUTE POINT....2544
DLGEXIT....2546
DLGFLUSH....2547
DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR....2549
DLGINIT, DLGINITWITHRESOURCEHANDLE....2551
DLGISDLGMESSAGE....2552
DLGMODAL, DLGMODALWITHPARENT....2555
DLGMODELESS....2557
DLGSENDCTRLMESSAGE....2561
DLGSET, DLGSETINT, DLGSETLOG, DLGSETCHAR....2562
DLGSETCTRLEVENTHANDLER....2565
DLGSETRETURN....2567
DLGSETSUB....2568
DLGSETTITLE....2572
DLGUNINIT....2573
DNUM....2574
DO....2575
DO Directive....2579
DO WHILE....2584
DOT_PRODUCT....2586
DOUBLE COMPLEX....2587

DOUBLE PRECISION....2588
DPROD....2589
DRAND, DRANDM....2590
DRANSET....2592
DREAL....2593
DSHIFTL....2594
DSHIFTR....2594
DTIME....2595
E to F....2596
ELEMENTAL....2596
ELLIPSE, ELLIPSE_W (W*32, W*64)....2597
ELSE....2600
ELSE Directive....2600
ELSEIF Directive....2600
ELSE IF....2600
ELSE WHERE....2600
ENCODE....2601
END....2603
END DO....2604
ENDIF Directive....2605
END IF....2605
ENDFILE....2605
END FORALL....2608
END INTERFACE....2608
END MAP....2608
END SELECT....2609
END STRUCTURE....2609
Derived Type (TYPE)....2617
END UNION....2623
END WHERE....2626
ENTRY....2627
EOF....2629

EOSHIFT....2632
EPSILON....2635
EQUIVALENCE....2636
ERF....2640
ERFC....2641
ERRSNS....2642
ETIME....2643
EXIT Statement....2645
EXIT Subroutine....2646
EXP....2647
EXPONENT....2649
EXTERNAL....2650
FDATE....2652
FGETC....2653
FIND....2654
FINDFILEQQ....2656
FIRSTPRIVATE....2657
FIXEDFORMLINESIZE....2657
FLOAT....2658
FLOODFILL, FLOODFILL_W (W*32, W*64)....2658
FLOODFILLRGB, FLOODFILLRGB_W (W*32, W*64)....2661
FLOOR....2663
FLUSH Directive....2664
FLUSH Statement....2666
FLUSH Subroutine....2666
FOCUSQQ (W*32, W*64)....2667
FOR_DESCRIPTOR_ASSIGN (W*32, W*64)....2668
FOR_GET_FPE....2672
FOR_RTL_FINISH_....2673
FOR_RTL_INIT_....2674
FOR_SET_FPE....2674
FOR_SET_REENTRANCY....2681

FORALL....2682
FORMAT....2685
FP_CLASS....2691
FPUTC....2692
FRACTION....2693
FREE....2694
FREEFORM....2695
FSEEK....2696
FSTAT....2697
FTELL, FTELLI8....2702
FULLPATHQQ....2703
FUNCTION....2705
G....2712
GERROR....2712
GETACTIVEQQ (W*32, W*64)....2714
GETARCINFO (W*32, W*64)....2714
GETARG....2716
GETBKCOLOR (W*32, W*64)....2718
GETBKCOLORRGB (W*32, W*64)....2719
GETC....2722
GETCHARQQ....2723
GETCOLOR (W*32, W*64)....2725
GETCOLORRGB (W*32, W*64)....2727
GET_COMMAND....2730
GET_COMMAND_ARGUMENT....2731
GETCONTROLFPQQ....2732
GETCURRENTPOSITION, GETCURRENTPOSITION_W (W*32, W*64)....2735
GETCWD....2737
GETDAT....2738
GETDRIVEDIRQQ....2740
GETDRIVESIZEQQ....2741

GETDRIVESQQ.....2744
GETENV.....2745
GET_ENVIRONMENT_VARIABLE.....2745
GETENVQQ.....2748
GETEXCEPTIONPTRSQQ (i32, i64em).....2751
GETEXITQQ (W*32, W*64).....2753
GETFILEINFOQQ.....2754
GETFILLMASK (W*32, W*64).....2759
GETFONTINFO (W*32, W*64).....2762
GETGID.....2764
GETGTEXTEXTENT (W*32, W*64).....2764
GETGTEXTROTATION (W*32, W*64).....2766
GETHWNDQQ (W*32, W*64).....2767
GETIMAGE, GETIMAGE_W.....2768
GETLASTERROR.....2769
GETLASTERRORQQ.....2770
GETLINESTYLE (W*32, W*64).....2772
GETLOG.....2774
GETPHYSCOORD (W*32, W*64).....2775
GETPID.....2777
GETPIXEL, GETPIXEL_W (W*32, W*64).....2777
GETPIXELRGB, GETPIXELRGB_W (W*32, W*64).....2779
GETPIXELS (W*32, W*64).....2781
GETPIXELSRGB (W*32, W*64).....2782
GETPOS, GETPOSI8.....2785
GETSTATUSFPQQ.....2785
GETSTRQQ.....2787
GETTEXTCOLOR (W*32, W*64).....2789
GETTEXTCOLORRGB (W*32, W*64).....2790
GETTEXTPOSITION (W*32, W*64).....2792
GETTEXTWINDOW (W*32, W*64).....2793
GETTIM.....2795

GETTIMEOFDAY....2796
GETUID....2796
GETUNITQQ (W*32, W*64)....2797
GETVIEWCOORD, GETVIEWCOORD_W (W*32, W*64)..2798
GETWINDOWCONFIG (W*32, W*64)....2799
GETWINDOWCOORD (W*32, W*64)....2804
GETWRITEMODE (W*32, W*64)....2805
GETWSIZEQQ (W*32, W*64)....2806
GMTIME....2808
GOTO - Assigned....2810
GOTO - Computed....2811
GOTO - Unconditional....2813
GRSTATUS (W*32, W*64)....2814
H to I....2819
HOSTNAM....2819
HUGE....2820
IACHAR....2820
IAND....2821
IARGC....2823
IBCHNG....2824
IBCLR....2825
IBITS....2826
IBSET....2827
ICHAR....2829
IDATE Intrinsic Procedure....2830
IDATE Portability Routine....2831
IDATE4....2832
IDENT....2833
IDFLOAT....2833
IEEE_CLASS....2834
IEEE_COPY_SIGN....2835
IEEE_GET_FLAG....2835

IEEE_GET_HALTING_MODE.....2836
IEEE_GET_ROUNDING_MODE.....2837
IEEE_GET_STATUS.....2838
IEEE_GET_UNDERFLOW_MODE.....2839
IEEE_IS_FINITE.....2839
IEEE_IS_NAN.....2840
IEEE_IS_NEGATIVE.....2841
IEEE_IS_NORMAL.....2841
IEEE_LOGB.....2842
IEEE_NEXT_AFTER.....2843
IEEE_REM.....2844
IEEE_RINT.....2844
IEEE_SCALB.....2845
IEEE_SELECTED_REAL_KIND.....2846
IEEE_SET_FLAG.....2847
IEEE_SET_HALTING_MODE.....2847
IEEE_SET_ROUNDING_MODE.....2848
IEEE_SET_STATUS.....2849
IEEE_SET_UNDERFLOW_MODE.....2850
IEEE_SUPPORT_DATATYPE.....2851
IEEE_SUPPORT_DENORMAL.....2852
IEEE_SUPPORT_DIVIDE.....2852
IEEE_SUPPORT_FLAG.....2853
IEEE_SUPPORT_HALTING.....2854
IEEE_SUPPORT_INF.....2854
IEEE_SUPPORT_IO.....2855
IEEE_SUPPORT_NAN.....2856
IEEE_SUPPORT_ROUNDING.....2856
IEEE_SUPPORT_SQRT.....2857
IEEE_SUPPORT_STANDARD.....2858
IEEE_SUPPORT_UNDERFLOW_CONTROL.....2859
IEEE_UNORDERED.....2860

IEEE_VALUE.....2860
IEEE_FLAGS.....2861
IEEE_HANDLER.....2867
IEOR.....2870
IERRNO.....2871
IF - Arithmetic.....2873
IF - Logical.....2875
IF Construct.....2876
IF Directive Construct.....2884
IF DEFINED Directive.....2887
IFIX.....2887
IFLOATI, IFLOATJ.....2887
ILEN.....2888
IMAGESIZE, IMAGESIZE_W (W*32, W*64).....2888
IMPLICIT.....2889
IMPORT.....2891
INCHARQQ (W*32, W*64).....2892
INCLUDE.....2895
INDEX.....2898
INITIALIZEFONTS (W*32, W*64).....2899
INITIALSETTINGS (W*32, W*64).....2900
INMAX.....2901
INQFOCUSQQ (W*32, W*64).....2902
INQUIRE.....2903
INSERTMENUQQ (W*32, W*64).....2907
INT.....2910
INTC.....2913
INT_PTR_KIND.....2914
INTEGER.....2915
INTEGER Directive.....2916
INTEGERTORGB (W*32, W*64).....2917
INTENT.....2919

INTERFACE....2923
INTERFACE TO....2926
INTRINSIC....2927
INUM....2929
IOR....2929
IPXFARGC....2931
IPXFCONST....2932
IPXFLENTRIM....2932
IPXFWEXITSTATUS (L*X, M*X)....2933
IPXFWSTOPSIG (L*X, M*X)....2935
IPXFWTERMSIG (L*X, M*X)....2936
IRAND, IRANDM....2936
IRANGET....2938
IRANSET....2938
ISATTY....2939
IS_IOSTAT_END....2939
IS_IOSTAT_EOR....2940
ISHA....2941
ISHC....2942
ISHFT....2943
ISHFTC....2945
ISHL....2947
ISNAN....2948
ITIME....2948
IVDEP....2949
IXOR....2951
J to L....2951
JABS....2951
JDATE....2952
JDATE4....2953
JNUM....2953
KILL....2954

KIND....2955
KNUM....2956
LASTPRIVATE....2957
LBOUND....2958
LCWRQQ....2959
LEADZ....2960
LEN....2961
LEN_TRIM....2962
LGE....2963
LGT....2964
LINETO, LINETO_W (W*32, W*64)....2966
LINETOAR (W*32, W*64)....2968
LINETOAREX (W*32, W*64)....2970
LLE....2973
LLT....2974
LNBLNK....2975
LOADIMAGE, LOADIMAGE_W (W*32, W*64)....2976
LOC....2977
%LOC....2978
LOG....2979
LOG10....2980
LOGICAL....2982
LOGICAL Function....2983
LONG....2983
LOOP COUNT....2984
LSHIFT....2986
LSTAT....2986
LTIME....2987
M to N....2989
MAKEDIRQQ....2989
MALLOC....2990
END MAP....2992

MASTER....2992
MATMUL....2993
MAX....2995
MAXEXPONENT....2997
MAXLOC....2998
MAXVAL....3002
MBCharLen....3004
MBConvertMBToUnicode....3005
MBConvertUnicodeToMB....3007
MBCurMax....3008
MBINCHARQQ....3009
MBINDEX....3010
MBJISToJMS, MBJMSToJIS....3011
MBLead....3012
MBLen....3013
MBLen_Trim....3014
MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE....3015
MBNext....3017
MBPrev....3018
MBSCAN....3019
MBStrLead....3020
MBVERIFY....3021
MCLOCK....3022
MEMORYTOUCH (i64 only)....3022
MEMREF_CONTROL (i64 only)....3023
MERGE....3024
MESSAGE....3026
MESSAGEBOXQQ (W*32, W*64)....3026
MIN....3028
MINEXPONENT....3030
MINLOC....3031
MINVAL....3035

MM_PREFETCH.....3037
MOD.....3040
MODIFYMENUFLAGSQQ (W*32, W*64).....3042
MODIFYMENUROUTINEQQ (W*32, W*64).....3043
MODIFYMENUSTRINGQQ (W*32, W*64).....3045
MODULE.....3047
MODULE PROCEDURE.....3053
MODULO.....3054
MOVE_ALLOC.....3055
MOVETO, MOVETO_W (W*32, W*64).....3058
MULT_HIGH (i64 only).....3060
MULT_HIGH_SIGNED (i64 only).....3061
MVBITS.....3062
NAMELIST.....3064
NARGS.....3066
NEAREST.....3069
NEW_LINE.....3070
NINT.....3071
NLSEnumCodepages.....3072
NLSEnumLocales.....3073
NLSFormatCurrency.....3074
NLSFormatDate.....3076
NLSFormatNumber.....3078
NLSFormatTime.....3079
NLSGetEnvironmentCodepage.....3081
NLSGetLocale.....3082
NLSGetLocaleInfo.....3083
NLSSetEnvironmentCodepage.....3095
NLSSetLocale.....3096
FREEFORM.....3098
NOOPTIMIZE.....3099
NOPREFETCH.....3100

NOSTRICT....3103
NOSWP (i64 only)....3105
NOT....3106
NOUNROLL....3108
NOVECTOR....3109
NULL....3110
NULLIFY....3112
O to P....3114
OBJCOMMENT....3114
OPEN....3115
OPTIONAL....3118
OPTIONS Statement....3122
NOOPTIMIZE....3123
OPTIONS Directive....3124
OR....3128
ORDERED....3129
OUTGTEXT (W*32, W*64)....3130
OUTTEXT (W*32, W*64)....3133
PACK Function....3134
PACK Directive....3136
PACKTIMEQQ....3138
PARALLEL....3139
PARALLEL ALWAYS....3142
PARALLEL ALWAYS....3143
PARALLEL DO....3145
PARALLEL SECTIONS....3146
PARALLEL WORKSHARE....3147
PARAMETER....3148
PASSDIRKEYSQQ (W*32, W*64)....3150
PAUSE....3156
PEEKCHARQQ....3158
PERROR....3159

PIE, PIE_W (W*32, W*64).....3160
POINTER - Fortran 95/90.....3163
POINTER - Integer.....3166
POLYBEZIER, POLYBEZIER_W (W*32, W*64).....3169
POLYBEZIERTO, POLYBEZIERTO_W (W*32, W*64).....3175
POLYGON, POLYGON_W (W*32, W*64).....3181
POLYLINEQQ (W*32, W*64).....3185
POPCNT.....3187
POPPAR.....3188
PRECISION.....3189
NOPREFETCH.....3189
PRESENT.....3192
PRINT.....3194
PRIVATE Statement.....3196
PRIVATE Clause.....3200
PRODUCT.....3201
PROGRAM.....3203
PROTECTED.....3205
PSECT.....3207
PUBLIC.....3208
PURE.....3212
PUTC.....3215
PUTIMAGE, PUTIMAGE_W (W*32, W*64).....3216
PXF(type)GET.....3220
PXF(type)SET.....3221
PXFA(type)GET.....3223
PXFA(type)SET.....3224
PXFACCESS.....3226
PXFALARM.....3227
PXFCALLSUBHANDLE.....3228
PXFCFGETISPEED (L*X, M*X).....3229
PXFCFGETOSPEED (L*X, M*X).....3229

PXFCFSETISPEED (L*X, M*X).....3230
PXFCFSETOSPEED (L*X, M*X).....3231
PXFCHDIR.....3231
PXFCHMOD.....3232
PXFCHOWN (L*X, M*X).....3233
PXFCLEARENV.....3233
PXFCLOSE.....3234
PXFCLOSEDIR.....3234
PXFCONST.....3235
PXFCREAT.....3236
PXFCTERMID.....3237
PXFDUP, PXFDUP2.....3237
PXFE(type)GET.....3238
PXFE(type)SET.....3239
PXFEXECV.....3241
PXFEXECVE.....3242
PXFEXECVP.....3243
PXFEXIT, PXFFASTEXIT.....3244
PXFFCNTL (L*X, M*X).....3245
PXFFDOPEN.....3248
PXFFFLUSH.....3250
PXFFGETC.....3250
PXFFILENO.....3251
PXFFORK (L*X, M*X).....3252
PXFFPATHCONF.....3254
PXFFPUTC.....3256
PXFFSEEK.....3257
PXFFSTAT.....3258
PXFFTELL.....3258
PXFGETARG.....3259
PXFGETATTY.....3260
PXFGETC.....3260

PXFGETCWD.....3261
PXFGETEGID (L*X, M*X).....3261
PXFGETENV.....3262
PXFGETEUID (L*X, M*X).....3263
PXFGETGID (L*X, M*X).....3263
PXFGETGRGID (L*X, M*X).....3264
PXFGETGRNAM (L*X, M*X).....3264
PXFGETGROUPS (L*X, M*X).....3265
PXFGETLOGIN.....3268
PXFGETPGRP (L*X, M*X).....3269
PXFGETPID.....3269
PXFGETPPID.....3271
PXFGETPWNAM (L*X, M*X).....3272
PXFGETPWUID (L*X, M*X).....3273
PXFGETSUBHANDLE.....3274
PXFGETUID (L*X, M*X).....3275
PXFISBLK.....3275
PXFISCHR.....3276
PXFISCONST.....3276
PXFISDIR.....3277
PXFISFIFO.....3278
PXFISREG.....3278
PXFKILL.....3279
PXFLINK.....3280
PXFLOCALTIME.....3281
PXFLSEEK.....3282
PXFMKDIR.....3283
PXFMKFIFO (L*X, M*X).....3284
PXFOPEN.....3284
PXFOPENDIR.....3288
PXFPATHCONF.....3289
PXFPAUSE.....3291

PXFPIPE....3292
PXFPOSIXIO....3292
PXFPUTC....3293
PXFREAD....3294
PXFREADDIR....3295
PXFRENAME....3295
PXFREWINDDIR....3296
PXFRMDIR....3297
PXFSETENV....3297
PXFSETGID (L*X, M*X)....3299
PXFSETPGID (L*X, M*X)....3300
PXFSETSID (L*X, M*X)....3301
PXFSETUID (L*X, M*X)....3301
PXFSIGACTION....3302
PXFSIGADDSET (L*X, M*X)....3303
PXFSIGDELSET (L*X, M*X)....3304
PXFSIGEMPTYSET (L*X, M*X)....3305
PXFSIGFILLSET (L*X, M*X)....3306
PXFSIGISMEMBER (L*X, M*X)....3306
PXFSIGPENDING (L*X, M*X)....3307
PXFSIGPROCMASK (L*X, M*X)....3308
PXFSIGSUSPEND (L*X, M*X)....3309
PXFSLEEP....3310
PXFSTAT....3310
PXFSTRUCTCOPY....3311
PXFSTRUCTCREATE....3312
PXFSTRUCTFREE....3318
PXFSYSCONF....3319
PXFTCDRAIN (L*X, M*X)....3322
PXFTCFLOW (L*X, M*X)....3322
PXFTCFLUSH (L*X, M*X)....3323
PXFTCGETATTR (L*X, M*X)....3324

PXFTCGETPGRP (L*X, M*X).....3325
PXFTCSENDBREAK (L*X, M*X).....3326
PXFTCSETATTR (L*X, M*X).....3326
PXFTCSETPGRP (L*X, M*X).....3327
PXFTIME.....3328
PXFTIMES.....3329
PXFTTYNAM (L*X, M*X).....3333
PXFUCOMPARE.....3333
PXFUMASK.....3334
PXFUNAME.....3334
PXFUNLINK.....3335
PXFUTIME.....3335
PXFWAIT (L*X, M*X).....3336
PXFWAITPID (L*X, M*X).....3338
PXFWIFEXITED (L*X, M*X).....3340
PXFWIFSIGNALED (L*X, M*X).....3342
PXFWIFSTOPPED (L*X, M*X).....3342
PXFWRITE.....3343
Q to R.....3344
QCMPLX.....3344
QEXT.....3345
QFLOAT.....3346
QNUM.....3347
QRANSET.....3347
QREAL.....3348
QSORT.....3348
RADIX.....3350
RAISEQQ.....3351
RAN.....3352
RAND, RANDOM.....3353
RANDOM.....3355
RANDOM_NUMBER.....3357

RANDOM_SEED....3360
RANDU....3362
RANF....3363
RANGE....3363
RANGET....3364
RANSET....3364
READ....3365
REAL Statement....3368
REAL Directive....3370
REAL Function....3371
RECORD....3373
RECTANGLE, RECTANGLE_W (W*32, W*64)....3374
RECURSIVE....3377
REDUCTION....3378
%REF....3381
REGISTERMOUSEEVENT (W*32, W*64)....3383
REMAPALLPALETTERGB, REMAPPALETTERGB (W*32, W*64)....3384
RENAME....3387
RENAMEFILEQQ....3388
REPEAT....3390
RESHAPE....3390
RESULT....3392
RETURN....3394
REWIND....3397
REWRITE....3398
RGBTOINTEGER (W*32, W*64)....3399
RINDEX....3401
RNUM....3402
RRSPACING....3402
RSHIFT....3403
RTC....3403

RUNQQ....3404
S....3405
SAVE....3405
SAVEIMAGE, SAVEIMAGE_W (W*32, W*64)....3408
SCALE....3409
SCAN....3410
SCANENV....3412
SCROLLTEXTWINDOW (W*32, W*64)....3412
SCWRQQ....3415
SECNDS Intrinsic Procedure....3416
SECNDS Portability Routine....3417
SECTIONS....3418
SEED....3420
END SELECT....3421
SELECTED_CHAR_KIND....3422
SELECTED_INT_KIND....3422
SELECTED_REAL_KIND....3423
SEQUENCE....3425
SETACTIVEQQ (W*32, W*64)....3427
SETBKCOLOR (W*32, W*64)....3428
SETBKCOLORRGB (W*32, W*64)....3429
SETCLIPRGN (W*32, W*64)....3431
SETCOLOR (W*32, W*64)....3434
SETCOLORRGB (W*32, W*64)....3436
SETCONTROLFPQQ....3438
SETDAT....3441
SETENVQQ....3442
SETERRORMODEQQ....3444
SETEXITQQ....3445
SET_EXPONENT....3448
SETFILEACCESSQQ....3448
SETFILETIMEQQ....3450

SETFILLMASK (W*32, W*64).....3451
SETFONT (W*32, W*64).....3455
SETGTEXTROTATION (W*32, W*64).....3460
SETLINESTYLE (W*32, W*64).....3462
SETMESSAGEQQ (W*32, W*64).....3464
SETMOUSECURSOR (W*32, W*64).....3466
SETPIXEL, SETPIXEL_W (W*32, W*64).....3469
SETPIXELRGB, SETPIXELRGB_W (W*32, W*64).....3470
SETPIXELS (W*32, W*64).....3473
SETPIXELSRGB (W*32, W*64).....3474
SETTEXTCOLOR (W*32, W*64).....3477
SETTEXTCOLORRGB (W*32, W*64).....3478
SETTEXTCURSOR (W*32, W*64).....3480
SETTEXTPOSITION (W*32, W*64).....3483
SETTEXTWINDOW (W*32, W*64).....3484
SETTIM.....3485
SETVIEWORG (W*32, W*64).....3487
SETVIEWPORT.....3488
SETWINDOW (W*32, W*64).....3489
SETWINDOWCONFIG (W*32, W*64).....3491
SETWINDOWMENUQQ (W*32, W*64).....3497
SETWRITEMODE (W*32, W*64).....3498
SETWSIZEQQ (W*32, W*64).....3502
SHAPE.....3504
SHARED.....3507
SHIFTL.....3507
SHIFTR.....3508
SHORT.....3508
SIGN.....3509
SIN.....3511
SIND.....3512
SINH.....3513

SIGNAL....3513
SIGNALQQ....3516
SINGLE....3520
SIZE....3521
SIZEOF....3522
SLEEP....3523
SLEEPQQ....3524
SNGL....3525
SORTQQ....3525
SPACING....3527
SPLITPATHQQ....3528
SPORT_CANCEL_IO....3530
SPORT_CONNECT....3531
SPORT_CONNECT_EX....3533
SPORT_GET_HANDLE....3535
SPORT_GET_STATE....3536
SPORT_GET_STATE_EX....3537
SPORT_GET_TIMEOUTS....3540
SPORT_PEEK_DATA....3542
SPORT_PEEK_LINE....3543
SPORT_PURGE....3544
SPORT_READ_DATA....3545
SPORT_READ_LINE....3546
SPORT_RELEASE....3548
SPORT_SET_STATE....3549
SPORT_SET_STATE_EX....3550
SPORT_SET_TIMEOUTS....3553
SPORT_SHOW_STATE....3554
SPORT_SPECIAL_FUNC....3556
SPORT_WRITE_DATA....3557
SPORT_WRITE_LINE....3558
SPREAD....3559

SQRT....3561
SRAND....3562
SSWRQQ....3564
STAT....3564
Statement Function....3569
STATIC....3572
STOP....3575
NOSTRICT....3577
END STRUCTURE....3579
SUBROUTINE....3586
SUM....3590
NOSWP (i64 only)....3592
SYSTEM....3593
SYSTEM_CLOCK....3595
SYSTEMQQ....3596
T to Z....3598
TAN....3598
TAND....3599
TANH....3599
TARGET....3600
TASK....3602
TASKWAIT....3606
THREADPRIVATE....3607
TIME Intrinsic Procedure....3608
TIME Portability Routine....3609
TIMEF....3611
TINY....3612
TRACEBACKQQ....3612
TRAILZ....3615
TRANSFER....3616
TRANSPOSE....3618
TRIM....3619

TTYNAM.....3620
Derived Type (TYPE).....3620
Type Declarations.....3626
DEFINE.....3632
END UNION.....3634
UNLINK.....3637
UNPACK.....3638
UNPACKTIMEQQ.....3640
UNREGISTERMOUSEEVENT (W*32, W*64).....3641
NOUNROLL.....3643
UNROLL_AND_JAM.....3644
USE.....3645
%VAL.....3651
VALUE.....3653
VECTOR ALIGNED.....3654
NOVECTOR.....3655
VECTOR TEMPORAL and VECTOR NONTEMPORAL (i32, i64em).....3657
VECTOR TEMPORAL and VECTOR NONTEMPORAL (i32, i64em).....3658
VECTOR ALIGNED.....3659
VERIFY.....3660
VIRTUAL.....3661
VOLATILE.....3661
WAIT.....3663
WAITONMOUSEEVENT (W*32, W*64).....3664
WHERE.....3666
WORKSHARE.....3670
WRAPON (W*32, W*64).....3671
WRITE.....3673
XOR.....3676
ZEXT.....3676

Chapter 64: Glossary

Glossary A....3679
Glossary B....3682
Glossary C....3683
Glossary D....3685
Glossary E....3689
Glossary F....3690
Glossary G....3692
Glossary H....3692
Glossary I....3693
Glossary K....3695
Glossary L....3695
Glossary M....3697
Glossary N....3698
Glossary O....3699
Glossary P....3700
Glossary Q....3702
Glossary R....3702
Glossary S....3704
Glossary T....3708
Glossary U....3709
Glossary V....3710
Glossary W....3710
Glossary Z....3711

Legal Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL(R) PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

* Other names and brands may be claimed as the property of others.

Getting Help and Support

The Intel® Fortran Compiler lets you build and optimize Fortran applications for the Linux* OS (operating system).

For more information about the compiler features and other components, see your *Release Notes*.

This documentation assumes that you are familiar with the Fortran programming language and with your processor's architecture. You should also be familiar with the host computer's operating system.

Product Website and Support

For general information on support for Intel software products, visit the Intel web site http://www.intel.com/software/products/

At this site, you will find comprehensive product information, including:

- Links to each product, where you will find technical information such as white papers and articles
- Links to user forums
- Links to news and events

To find technical support information, to register your product, or to contact Intel, please visit: http://www.intel.com/software/products/support/

For additional information, see the Technical Support section of your Release Notes.

System Requirements

For detailed information on system requirements, see the Release Notes.

Introduction

1

Introducing the Intel® Fortran Compiler

The Intel® Fortran Compiler can generate code for IA-32, Intel® 64, or IA-64 architecture applications on any Intel®-based Linux* system. IA-32 architecture applications (32-bit) can run on all Intel®-based Linux systems. Intel® 64 architecture applications and IA-64 architecture applications can run only on Intel® 64 architecture-based or IA-64 architecture-based Linux systems. You can use the compiler on the command line.

You can find further information in the following documents:

- Building Applications
- Compiler Options
- Optimizing Applications
- Floating-point Operations
- Language Reference

Notational Conventions

Information in this documentation applies to all supported operating systems and architectures unless otherwise specified.

This documentation uses the following conventions:

- Notational Conventions
- Platform Labels

Notational Conventions

THIS TYPE	Indicates statements, data types, directives, and other language keywords. Examples of statement keywords are WRITE, INTEGER, DO, and OPEN.

this type	Indicates command-line or option arguments, new terms, or emphasized text. Most new terms are defined in the Glossary.
`This type`	Indicates a code example.
`This type`	Indicates what you type as input.
This type	Indicates menu names, menu items, button names, dialog window names, and other user-interface items.
File>Open	Menu names and menu items joined by a greater than (>) sign indicate a sequence of actions. For example, "Click **File>Open**" indicates that in the **File** menu, click **Open** to perform this action.
{value | value}	Indicates a choice of items or values. You can usually only choose one of the values in the braces.
[*item*]	Indicates items that are optional. Brackets are also used in code examples to show arrays.
item [, *item*]...	Indicates that the item preceding the ellipsis (three dots) can be repeated. In some code examples, a horizontal ellipsis means that not all of the statements are shown.
Windows* OS Windows operating system	These terms refer to all supported Microsoft* Windows* operating systems.
Linux* OS Linux operating system	These terms refer to all supported Linux* operating systems.
Mac OS* X Mac OS X operating system	These terms refer to Intel©-based systems running the Mac OS* X operating system.

Microsoft Windows XP*	An asterisk at the end of a word or name indicates it is a third-party product trademark.
compiler option	This term refers to Windows* OS options, Linux* OS options, or MAC OS* X options that can be used on the compiler command line.

Conventions Used in *Compiler Options*

`/option` or `-option`	A slash before an option name indicates the option is available on Windows OS. A dash before an option name indicates the option is available on Linux OS* and Mac OS* X systems. For example: Windows option: `/fast` Linux and Mac OS X option: `-fast` Note: If an option is available on Windows* OS, Linux* OS, and Mac OS* X systems, no slash or dash appears in the general description of the option. The slash and dash will only appear where the option syntax is described.
`/option:argument` or `-option argument`	Indicates that an option requires a argument (parameter). For example, you must specify an argument for the following options: Windows OS option: `/Qdiag-error-limit:`*n* Linux OS and Mac OS X option: `-diag-error-limit` *n*
`/option:`*keyword* or `-option` *keyword*	Indicates that an option requires one of the *keyword* values.
`/option`[:*keyword*] or -option [*keyword*]	Indicates that the option can be used alone or with an optional keyword.

`option[n]` or `option[:n]` or `option[=n]`	Indicates that the option can be used alone or with an optional value; for example, in `/Qfnalign[:n]` and `-falign-functions[=n]`, the *n* can be omitted or a valid value can be specified for *n*.
`option[-]`	Indicates that a trailing hyphen disables the option; for example, `/Qglobal_hoist-` disables the Windows OS option `/Qglobal_hoist`.
`[no]option` or `[no-]option`	Indicates that "no" or "no-" preceding an option disables the option. For example: In the Windows OS option `/[no]traceback`, `/traceback` enables the option, while `/notraceback` disables it. In the Linux OS and Mac OS X option `-[no-]global_hoist`, `-global_hoist` enables the option, while `-no-global_hoist` disables it. In some options, the "no" appears later in the option name; for example, `-fno-alias` disables the `-falias` option.

Conventions Used in *Language Reference*

This color	Indicates extensions to the Fortran 95 Standard. These extensions may or may not be implemented by other compilers that conform to the language standard.
Intel Fortran	This term refers to the name of the common compiler language supported by the Intel® Fortran Compiler.
Fortran	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 95 and 90, and Intel Fortran.

Fortran 95/90	This term refers to language information that is common to ANSI FORTRAN 77, ANSI/ISO Fortran 95, ANSI/ISO Fortran 90, and Intel Fortran.
Fortran 95	This term refers to language features specific to ANSI/ISO Fortran 95.
integer	This term refers to the INTEGER(KIND=1), INTEGER(KIND=2), INTEGER (INTEGER(KIND=4)), and INTEGER(KIND=8) data types as a group.
real	This term refers to the REAL (REAL(KIND=4)), DOUBLE PRECISION (REAL(KIND=8)), and REAL(KIND=16) data types as a group.
REAL	This term refers to the default data type of objects declared to be REAL. REAL is equivalent to REAL(KIND=4), unless a compiler option specifies otherwise.
complex	This term refers to the COMPLEX (COMPLEX(KIND=4)), DOUBLE COMPLEX (COMPLEX(KIND=8)), and COMPLEX(KIND=16) data types as a group.
logical	This term refers to the LOGICAL(KIND=1), LOGICAL(KIND=2), LOGICAL (LOGICAL(KIND=4)), and LOGICAL(KIND=8) data types as a group.
Compatibility	This term introduces a list of the projects or libraries that are compatible with the library routine.
< Tab>	This symbol indicates a nonprinting tab character.

^	This symbol indicates a nonprinting blank character.

Platform Labels

A platform is a combination of operating system and central processing unit (CPU) that provides a distinct environment in which to use a product (in this case, a computer language). An example of a platform is Microsoft* Windows* XP on processors using IA-32 architecture.

In this documentation, information applies to all supported platforms unless it is otherwise labeled for a specific platform (or platforms).

These labels may be used to identify specific platforms:

L*X	Applies to Linux* OS on processors using IA-32 architecture, Intel® 64 architecture, and IA-64 architecture.
L*X32	Applies to Linux* OS on processors using IA-32 architecture and Intel® 64 architecture.
L*X64	Applies to Linux OS on processors using IA-64 architecture.
M*X	Applies to Apple* Mac OS* X on processors using IA-32 architecture and Intel® 64 architecture.
M*X32	Applies to Apple* Mac OS* X on processors using IA-32 architecture.
M*X64	Applies to Apple* Mac OS* X on processors using Intel® 64 architecture.
W*32	Applies to Microsoft Windows* 2000, Windows XP, and Windows Server 2003 on processors using IA-32 architecture and Intel® 64 architecture. For a complete list of supported Windows* operating systems, see your Release Notes.

W*64	Applies to Microsoft Windows* XP operating systems on IA-64 architecture.
i32	Applies to 32-bit operating systems on IA-32 architecture.
i64em	Applies to 32-bit operating systems on Intel® 64 architecture.
i64	Applies to 64-bit operating systems on IA-64 architecture.

Related Information

Tutorial information

The following commercially published documents provide reference or tutorial information on Fortran 2003, Fortran 95, and Fortran 90:

- *Introduction to Programming with Fortran with coverage of Fortran 90, 95, 2003 and 77*, by I.D. Chivers and J. Sleightholme; published by Springer, ISBN 9781846280535
- *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*, by Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T., published by Springer Verlag, ISBN: 9781846283789
- *Fortran 95/2003 For Scientists and Engineers*, by Chapman S.J., published by McGraw- Hill. ISBN ISBN 0073191574
- *Fortran 95/2003 Explained*, by Metcalf M., Reid J. and Cohen M., 2004, published by Oxford University Press. ISBN 0-19-852693-8
- *Object Oriented Programming via Fortran 90/95*, by Akin E., published by Cambridge University Press, ISBN 0-521-52408-3
- *Introducing Fortran 95*, by Chivers I.D., Sleightholme J., published by Springer Verlag, ISBN 185233276X
- *Fortran 95 Handbook*, by Adams J.C., Brainerd W.S., Martin J.T, Smith B.T., and Wagener J.L, published by MIT, ISBN 0-262-51096-0
- *Fortran 90/95 for Scientists and Engineers*, by S. J. Chapman; published by McGraw-Hill, ISBN 0-07-282575-8

- *Fortran 90/95 Explained*, by M. Metcalf and J. Reid; published by Oxford University Press, ISBN 0-19-851888-9

Intel does not endorse these books or recommend them over other books on the same subjects.

Standard and Specification Documents

The following copyrighted standard and specification documents provide descriptions of many of the features found in Intel® Fortran:

- American National Standard Programming Language FORTRAN, ANSI X3.9-1978
- American National Standard Programming Language Fortran 90, ANSI X3.198-1992
  This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539:1991 (E).
- American National Standard Programming Language Fortran 95, ANSI X3J3/96-007
  This Standard is equivalent to: International Standards Organization Programming Language Fortran, ISO/IEC 1539-1:1997 (E).
- International Standards Organization Information Technology - Programming Languages - Fortran, ISO/IEC 1539-1:2004 (E)
  This is the Fortran 2003 Standard.
- High Performance Fortran Language Specification, Version 1.1, Technical Report CRPC-TR-92225
- OpenMP Fortran Application Program Interface, Version 1.1, November 1999
- OpenMP Fortran Application Program Interface, Version 2.0, November 2000

Associated Intel Documents

The following Intel documents provide additional information about the Intel® Fortran Compiler, Intel® architecture, Intel® processors, or tools:

- *Using the Intel® License Manager for FLEXlm**
- *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel Corporation
- *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*, Intel Corporation
- *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, Intel Corporation

- *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, Intel Corporation
- *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*, Intel Corporation
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*
- *Intel® Itanium® Architecture Software Developer's Manual - Volume 1: Application Architecture, Revision 2.2*
- *Intel® Itanium® Architecture Software Developer's Manual - Volume 2: System Architecture, Revision 2.2*
- *Intel® Itanium® Architecture Software Developer's Manual - Volume 3: Instruction Set Reference, Revision 2.2*
- *Intel® Processor Identification with the CPUID Instruction*, Intel Corporation, doc. number 241618
- *IA-64 Architecture Assembler User's Guide*
- *IA-64 Architecture Assembly Language Reference Guide*

Most Intel documents can be found at the Intel web site http://www.intel.com/software/products/

Optimization and Vectorization Terminology and Technology

The following documents provide details on basic optimization and vectorization terminology and technology:

- *Intel® Architecture Optimization Reference Manual*
- *Dependence Analysis*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1997.
- *The Structure of Computers and Computation: Volume I*, David J. Kuck. John Wiley and Sons, New York, 1978.
- *Loop Transformations for Restructuring Compilers: The Foundations*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1993.
- *Loop parallelization*, Utpal Banerjee (A Book Series on Loop Transformations for Restructuring Compilers). Kluwer Academic Publishers. 1994.
- *High Performance Compilers for Parallel Computers*, Michael J. Wolfe. Addison-Wesley, Redwood City. 1996.

- *Supercompilers for Parallel and Vector Computers*, H. Zima. ACM Press, New York, 1990.
- *An Auto-vectorizing Compiler for the Intel© Architecture*, Aart Bik, Paul Grey, Milind Girkar, and Xinmin Tian. Submitted for publication
- *Efficient Exploitation of Parallelism on Pentium® III and Pentium© 4 Processor-Based Systems*, Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian.
- *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*, A.J.C. Bik. Intel Press, June, 2004.
- *Multi-Core Programming: Increasing Performance through Software Multithreading*, Shameem Akhter and Jason Roberts. Intel Press, April, 2006.

Additional Training

For additional technical product information including white papers about Intel compilers, open the page associated with your product at http://www.intel.com/software/products/

Part

I

Building Applications

Topics:

- Overview: Building Applications
- Introduction: Basic Concepts
- Building Applications from the Command Line
- Input and Output Files
- Setting Environment Variables
- Using Compiler Options
- Preprocessing
- Using Configuration Files and Response Files
- Debugging
- Data and I/O
- Structuring Your Program
- Programming with Mixed Languages
- Using Libraries
- Error Handling
- Portability Considerations
- Troubleshooting
- Reference Information

Overview: Building Applications 2

Welcome to the Intel® Fortran Compiler.

This Building Applications document explains how to use the Intel® Compiler to build applications on Linux*, Windows*, and Mac OS* X operating systems. Intel® Fortran provides you with a variety of alternatives for building applications. Depending on your needs, you can build your source code into several types of programs and libraries using the command line. Additionally, on Windows systems, you can use the Microsoft Visual Studio* integrated development environment (IDE) and on Mac OS X operating systems, you can use the Xcode* IDE to build your application.

The discussions in this document often contain content that applies generally to all supported operating systems; however, where the expected behavior is significantly different on a specific OS, the appropriate behavior is listed separately.

In general, the compiler features and options supported on Linux OS using IA-32 architecture or Intel® 64 architecture are also supported on Intel®-based systems running Mac OS X. For more detailed information about support for specific operating systems, refer to the appropriate option in the Compiler Options reference or the Release Notes.

Introduction: Basic Concepts 3

Choosing Your Development Environment

Depending on your operating system, you can build programs from the command line and/or from an IDE such as Microsoft Visual Studio* (Windows* OS) or Xcode* (Mac OS* X).

An IDE offers a number of ways to simplify the task of compiling and linking programs. It provides a default text editor. You can also use your favorite text editor outside the integrated development environment.

Because software development is an iterative process, it is important to be able to move quickly and efficiently to various locations in your source code. If you use an IDE to build your programs, you can display both the description of the error message and the relevant source code directly from the displayed error messages.

When you build programs from the command line, you may have more control of the build tools. If you choose to, you can customize how your program is built by your selection of compiler and linker options. Compiler options are described in the Compiler Options Reference.

See Also

- Introduction: Basic Concepts
- Invoking the Intel® Fortran Compiler
- Using the Compiler and Linker from the Command Line

Invoking the Intel® Fortran Compiler

The command to invoke the compiler is `ifort`.

Requirements Before Using the Command Line

On Linux* and Mac OS* X operating systems, you need to set some environment variables to specify locations for the various components prior to using the command line. The Intel Fortran Compiler installation includes a shell script that you can run to set environment variables. For more information, see Using the ifortvars File to Specify Location of Components.

On Windows* operating systems, you typically do not need to set any environment variables prior to using the command line. Each of the Intel® Fortran Compiler variations has its own Intel Compiler command-line window, available from the Intel Fortran program folder. This window has the appropriate environment variables already set for the command-line environment.

Using the ifort Command from the Command Line

Use the `ifort` command either on a command line or in a makefile to invoke the Intel Fortran compiler. The syntax is:

`ifort [options]input_file(s)`

For a complete listing of compiler options, see the Compiler Options reference.

You can specify more than one input_file, using a space as a delimiter. See Understanding Input File Extensions.

For more information on ifort syntax, see Syntax for the ifort Command.

NOTE. For Windows and Mac OS* X systems, you can also use the compiler from within the integrated development environment.

You can use the command-line window to invoke the Intel Fortran Compiler in a number of ways, detailed below.

Using makefiles from the Command Line

Use makefiles to specify a number of files with various paths and to save this information for multiple compilations. For more information on using makefiles, see Using Makefiles to Compile Your Application.

Using the devenv command from the Command Line (Windows only)

Use `devenv` to set various options for the integrated development environment (IDE) as well as build, clean, and debug projects from the command line. For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio* documentation.

Using a .bat file from the Command Line

Use a .bat file to consistently execute the compiler with a desired set of options. This spares you retyping the command each time you need to recompile.

Default Tools

The default tools are summarized in the table below.

Tool	Default	Provided with Intel® Fortran Compiler?
Assembler for IA-32 architecture-based applications and Intel® 64 architecture-based applications	MASM* (Windows OS)	No
	operating system assembler, as (Linux OS and Mac OS X)	No
Assembler for IA-64 architecture-based applications	`ias`	Yes
Linker	Microsoft* linker (Windows OS)	No
	System linker, ld(1) (Linux OS and Mac OS X)	No

You can specify alternative tools and locations for preprocessing, compilation, assembly, and linking.

Assembler

By default, the compiler generates an object file directly without calling the assembler. However, if you need to use specific assembly input files and then link them with the rest of your project, you can use an assembler for these files.

IA-32 architecture-based applications

Use any 32-bit assembler. For Windows, you can use the Microsoft Macro Assembler* (MASM), version 6.15 or higher, to link assembly language files with the object files generated by the compiler.

Intel® 64 architecture-based applications

For Windows systems, use the MASM provided on the Microsoft SDK. For Linux OS and Mac OS X systems, use the operating system assembler, `as`.

IA-64 architecture-based applications

Use the assembler, `ias`. The following example compiles a Fortran file to an assembly language file, which you can modify as desired. The assembler is then used to create an object file.

Use the `-S` (Linux) or `/asmfile:file.asm` (Windows) option to generate an assembly code file.

- The following command line on Linux OS and Mac OS X generates the assembly code file, `file.s`:

  ```
  ifort -S -c file.f
  ```

- The following command line on Windows OS generates the assembly code for `file.asm`:

  ```
  ifort /asmfile:file /c file.f
  ```

To assemble the file just produced, call the IA-64 architecture assembler.

- The following is the Linux OS command line:

  ```
  ias -Nso -p32 -o file.o file.s
  ```

- The following is the Windows OS command line:

  ```
  ias /Nso /p32 /o file.obj file.asm
  ```

  where the following assembler options are used:

  - Nso suppresses the sign-on message
  - p32 enables defining 32-bit elements as relocatable data elements; kept for backward compatibility
  - The file specified by the `o` option indicates the output object file name

The above `ias` command generates an object file, which you can link with the object file of the project.

Linker

On Linux OS and Mac OS X, the compiler calls the system linker, `ld(1)`, to produce an executable file from the object file.

On Windows OS, the compiler calls the Microsoft linker, `link`, to produce an executable file from the object files. The linker searches the path specified in the environment variable `LIB` to find any library files.

Specifying Alternative Tools and Locations

The Intel® Fortran Compiler lets you specify alternatives to default tools and locations for preprocessing, compilation, assembly, and linking. In addition, you can invoke options specific to the alternate tools on the command line. This functionality is provided by the `-Qlocation` or `/Qlocation` and `-Qoption` or `/Qoption` options.

For more information see the following topics:

- Qlocation compiler option
- Qoption compiler option

Compilation Phases

When invoked, the compiler driver determines which compilation phases to perform based on the extension of the source filename and on the compilation options specified in the command line.

The table that follows shows the compilation phases and the software that controls each phase.

Phase	Software	Architecture (IA-32, Intel® 64, or IA-64)
Preprocess (optional)	`fpp`	All
Compile	`fortcom`	All

Phase	Software	Architecture (IA-32, Intel® 64, or IA-64)
Assemble	IAS or MASM (Windows OS)	IAS for IA-64 architecture based applications;Microsoft Macro Assembler* (MASM) can be used for IA-32 architecture based applications. See Default Tools for more information.
	as or ias (Linux OS)	as for IA-32 architecture-based applications and Intel® 64 architecture-based applications; ias for IA-64 architecture-based applications
Link	LINK (Windows OS)	All
	ld(1) (Linux OS and Mac OS X)	All

By default, the compiler driver performs the compile and link phases to produce the executable file.

The compiler driver passes object files and any unrecognized filename to the linker. The linker then determines whether the file is an object file or a library. For Linux OS and Mac OS X, the linker can also determine whether the file is a shared library (.so).

The compiler driver handles all types of input files correctly. Therefore, it can be used to invoke any phase of compilation.

The compiler processes Fortran language source and generates object files. You decide the input and output by setting options when you run the compiler.

When invoked, the compiler determines which compilation phases to perform based on the extension of the source filename and on the compilation options specified in the command line.

Compiling and Linking for Optimization

By default, all Fortran source files are separately compiled into individual object files.

If you want to allow full interprocedural optimizations to occur, you must use the -ipo (Linux OS and Mac OS X) or /Qipo (Windows OS) option.

By default, compilation is done with `-O2` (Linux OS and Mac OS X) or `/O2` (Windows). If you want to see if your code will benefit from some added optimizations, use `O3`. These aggressive optimizations may or may not improve your code speed.

For complete information about optimization, see Compiler Optimizations Overview in Optimizing Applications.

Compiling and Linking Multithread Programs

To build a multithread application that uses the Fortran run-time libraries, specify the `-threads` (Linux* OS and Mac OS* X) or `/threads` (Windows* OS) compiler option from the command line. For Windows systems, you can use also use the Microsoft integrated development environment (IDE), as described later in this topic.

You must also link with the correct library files.

The following applies to Linux OS and Mac OS X:

To create statically linked multithread programs, link with the static library named libifcoremt.a. To use shared libraries, link your application with libifcoremd.so (Linux OS) or libifcoremd.dylib (Mac OS X).

The following applies to Windows OS:

To create statically linked multithread programs, link with the re-entrant support library LIBIFCOREMT.LIB. To use shared libraries, use the shared LIBIFCOREMD.DLL library, which also re-entrant, and is referenced by linking your application with the LIBIFCOREMD.LIB import library.

Programs built with LIBIFCOREMT.LIB do not share Fortran run-time library code or data with any dynamic-link libraries they call. You must link with LIBIFCOREMD.LIB if you plan to call a DLL.

Additional Notes for Windows OS:

- The `/threads` compiler option is automatically set when you specify a multithread application in the visual development environment.
- Specify the compiler options `/libs=dll` and /threads if you are using both multithreaded code and DLLs. You can use the `/libs=dll` and `/threads` options only with Fortran Console projects, not QuickWin applications.

To compile and link your multithread program from the command line:

1. Make sure your IA32ROOT or IA64ROOT (Linux OS and Mac OS X) or LIB (Windows) environment variable points to the directory containing your library files.
2. Compile and link the program with the `-threads` (Linux OS and Mac OS X) or `/threads` (Windows) compiler option.

For example:

`ifort -threads mythread.f90` (Linux OS and Mac OS X)

`ifort /threads mythread.f90` (Windows OS)

To compile and link your multithread program using the IDE (Windows OS):

1. Create a new project by clicking File > New > Project.
2. Click **Intel Fortran Projects** in the left pane to display the Intel Fortran project types. Choose the project type.
3. Add the file containing the source code to the project.
4. From the **Project** menu, select **Properties**.

   The **Property Pages** dialog box appears.
5. Choose the **Fortran** folder, **Libraries** category, and set the Runtime Library to Multithreaded or Multithread DLL (or their debug equivalents).
6. Create the executable file by choosing Build Solution from the **Build** menu.

What the Compiler Does by Default

By default, the compiler driver generates executable file(s) of the input file(s) and performs the following actions:

- Displays certain diagnostic messages, warnings, and errors.
- Performs default settings and optimizations, unless these options are overridden by specific options settings.
- Searches for source files in the current directory or in the directory path explicitly specified before a file name (for example, looks in "src" when the directory path is `src\test.f90`).
- Searches for include and module files in:
  - The directory path explicitly specified before a file name (for example, looks in "src" when the including source is specified as `src\test.f90`)
  - The current directory
  - The directory specified by using the `-module path` (Linux* OS and Mac OS* X) or `/module:path` (Windows* OS) option (for all module files)
  - The directory specified by using the `-I`*`dir`* ((Linux OS and Mac OS X) or `/I`*`dir`* (Windows OS) option (for module files referenced in USE statements and include files referenced in INCLUDE statements.)

  - For Windows OS, the include path specified by the INCLUDE environment variable (for all include or module files)
  - Any directory explicitly specified in any INCLUDE within an included file

- Passes options designated for linking as well as user-defined libraries to the linker. The linker searches for any library files in directories specified by the LIB variable, if they are not found in the current directory.

For unspecified options, the compiler uses default settings or takes no action.

You may want to use the `-assume keyword` (Linux OS and Mac OS X) or `/assume:keyword` (Windows OS) option to instruct the compiler to make certain assumptions. For example, `-assume buffered_io` tells the compiler to accumulate records in a buffer. For more information and the complete list of supported keywords, see the assume option reference page.

NOTE. On operating systems that support characters in Unicode* (multi-byte) format, the compiler will process file names containing Unicode* characters.

Generating Listing and Map Files

Compiler-generatedassembler output listings and linker-generated map files can help you understand the effects of compiler optimizations and see how your application is laid out in memory. They may also help you interpret the information provided in a stack trace at the time of an error.

How to Generate Assembler Output

When compiling from the command line, specify the `-S` (Linux* OS and Mac OS* X) or `/asmattr` option with one of its keyword (Windows* OS):

```
ifort -S file.f90 (Linux OS and Mac OS X)
ifort file.f90 /asmattr:all (Windows OS)
```

On Linux OS and Mac OS X, the resulting assembly file name has a .s suffix. On Windows OS, the resulting assembly file name has an .asm suffix.

Additionally, on Windows OS, you can use the Visual Studio integrated development environment:

1. Select Project>Properties.
2. Click the Fortran tab.

3. In the Output Files category, change the Assembler Output settings according to your needs. You can choose from a number of options such as No Listing, Assembly-only Listing, and Assembly, Machine Code and Source.

How to Generate a Link Map (.map) File

When compiling from the command line, specify the `-Xlinker` and `-M` options (Linux OS and Mac OS X) or the `/map` (Windows) option:

```
ifort file.f90 -Xlinker -M  (Linux OS and Mac OS X)
ifort file.f90 /map (Windows)
```

Additionally, on Windows systems, you can use the Visual Studio integrated development environment:

1. Select Project>Properties.
2. Click the Linker tab.
3. In the Debug category, select Generate Map File.

Saving Compiler Information in your Executable

If you want to save information about the compiler in your executable, use the `-sox` (Linux* OS) or `/Qsox` (Windows* OS) option. When you use this option, the following information is saved:

- compiler version number
- compiler options that were used to produce the executable

On Linux OS:

To view the information stored in string format in the object file, use the following command:

```
strings -a a.out|grep comment:
```

On Windows OS:

To view the linker directives stored in string format in the object file, use the following command:

```
link /dump /directives filename.obj
```

The `-?comment` linker directive displays the compiler version information.

To search your executable for compiler information, use the following command:

```
findstr "Compiler" filename.exe
```

This searches for any strings that have the substring "Compiler" in them.

Building Applications from the Command Line

4

Using the Compiler and Linker from the Command Line

The `ifort` command is used to compile and link your programs from the command line.

You can either compile and link your projects in one step with the `ifort` command, or compile them with `ifort` and then link them as a separate step.

In most cases, you will use a single `ifort` command to invoke the compiler and linker.

You can use the `ifort` command in either of two windows:

- Your own terminal window, in which you have set the appropriate environment variables by executing the file called `ifortvars.sh` or `ifortvars.csh` (Linux* OS and Mac OS* X) or `ifortvars.bat` (Windows* OS). This file sets the environment variables such as PATH. By default, the `ifortvars` file is installed in the `\bin` directory for your compiler. For more information, see Using the ifortvars File to Specify Location of Components.
- On Windows operating systems, the supplied Fortran command-line window in the Intel® Fortran program folder, in which the appropriate environment variables in `ifortvars.bat` are preset.

The `ifort` command invokes a *`driver program`* that is the actual user interface to the compiler and linker. It accepts a list of command options and file names and directs processing for each file.

The driver program does the following:

- Calls the Intel® Fortran Compiler to process Fortran files.
- Passes the linker options to the linker.
- Passes object files created by the compiler to the linker.
- Passes libraries to the linker.
- Calls the linker or librarian to create the executable or library file.

You can also use `ld` (Linux OS and Mac OS X) or `link` (Windows OS) to build libraries of object modules. These commands provide syntax instructions at the command line if you request it with the `/?` or `/help` option.

The`ifort` command automatically references the appropriate Intel Fortran Run-Time Libraries when it invokes the linker. Therefore, to link one or more object files created by the Intel Fortran compiler, you should use the `ifort` command instead of the `link` command.

Because the driver calls other software components, error messages may be returned by these other components. For instance, the linker may return a message if it cannot resolve a global reference. The `-watch` (Linux OS and Mac OS X) or `/watch` (Windows OS) command-line option can help clarify which component is generating the error.

NOTE. Windows systems support characters in Unicode* (multibyte) format; the compiler will process file names containing Unicode* characters.

Syntax for the ifort Command

Use the syntax below to invoke the Intel® Fortran Compiler from the command line:

```
ifort  [options]input_file(s)
```

An option is specified by one or more letters preceded by a hyphen (-) for Linux and Mac OS* X operating systems and a slash (/) for the Windows* operating system. (You can use a hyphen (-) instead of a slash for Windows OS, but this is not the preferred method.)

The following rules apply:

- Some options take arguments in the form of filenames, strings, letters, or numbers. Except where otherwise noted, you can enter a space between the option and its argument(s) or you can combine them. For a complete listing of compiler options, see the Compiler Options reference.
- You can specify more than one input_file , using a space as a delimiter. When a file is not in your path or working directory, specify the directory path before the file name. The filename extension specifies the type of file.

NOTE. Options on the command line apply to all files. For example, in the following command line, the `-c` and `-nowarn` options apply to both files `x.f` and `y.f`:

```
ifort -c x.f -nowarn y.f
```

- You cannot combine options with a single slash or hyphen, but must specify the slash or hyphen for each option specified. For example, this is correct: /1 /c

  But this is not: /1c
- Some compiler options are case-sensitive. For example, c and C are two different options.
- Options can take arguments in the form of filenames, strings, letters, and numbers. If a string includes spaces, it must be enclosed in quotation marks.

- All compiler options must precede the `-Xlinker` (Linux OS and Mac OS X) or `/link` (Windows OS) options. Options that appear following -Xlinker or /link are passed directly to the linker.
- Unless you specify certain options, the command line will both compile and link the files you specify. To compile without linking, specify the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option.
- You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.
- Compiler options remain in effect for the whole compilation unless overridden by a compiler directive.
- On Windows OS, certain options accept one or more keyword arguments following the option name. To specify multiple keywords, you typically specify the option multiple times. However, some options allow you to use comma-separated keywords. For example:

```
ifort /warn:usage,declarations test.f90
```

You can use an equals sign (=) instead of the colon:

```
ifort /warn=usage,declarations test.f90
```

Examples of the ifort Command

This topic provides some examples of valid `ifort` commands. It also shows various ways to compile and link source files.

Compiling and Linking a Single Source File

The following command compiles `x.for`, links, and creates an executable file. This command generates a temporary object file, which is deleted after linking:

```
ifort x.for
```

To specify a particular name for the executable file, specify the -o (Linux* OS and Mac OS* X) or `/exe` (Windows* OS) option:

```
ifort x.for -o myprog.out (Linux OS and Mac OS X)
ifort x.for /exe:myprog.exe (Windows OS)
```

Compiling, but not Linking, a Source File

The following command compiles `x.for` and generates the object file `x.o` (Linux OS and Mac OS X) or `x.obj` (Windows OS). The `c` option prevents linking (it does not link the object file into an executable file):

```
ifort -c x.for (Linux OS and Mac OS X)
ifort x.for /c (Windows OS)
```

The following command links `x.o` or `x.obj` into an executable file. This command automatically links with the default Intel Fortran libraries:

```
ifort x.o (Linux OS and Mac OS X)
ifort x.obj (Windows OS)
```

Compiling and Linking Multiple Fortran Source Files

The following command compiles `a.for`, `b.for`, and `c.for`. It creates three temporary object files, then links the object files into an executable file named `a.out` (on Linux OS and Mac OS X) or `a.exe` (Windows OS):

```
ifort a.for b.for c.for
```

When you use modules and compile multiple files, compile the source files that define modules *before* the files that reference the modules (in `USE` statements).

When you use a single `ifort` command, the order in which files are placed on the command line is significant. For example, if the free-form source file `moddef.f90` defines the modules referenced by the file `projmain.f90`, use the following command:

```
ifort moddef.f90 projmain.f90
```

Creating, Running, and Debugging an Executable Program

The example below shows a sample Fortran main program using free source form that uses a module and an external subprogram.

The function CALC_AVERAGE is contained in a separate file and depends on the module ARRAY_CALCULATOR for its interface block.

The USE statement accesses the module ARRAY_CALCULATOR. This module contains the function declaration for CALC_AVERAGE.

The 5-element array is passed to the function CALC_AVERAGE, which returns the value to the variable AVERAGE for printing.

The example is:

```
! File: main.f90
! This program calculates the average of five numbers
PROGRAM MAIN
USE ARRAY_CALCULATOR
REAL, DIMENSION(5) :: A = 0
REAL :: AVERAGE
PRINT *, 'Type five numbers: '
READ  (*,'(F10.3)') A

AVERAGE = CALC_AVERAGE(A)
PRINT *, 'Average of the five numbers is: ', AVERAGE
END PROGRAM MAIN
```

The example below shows the module referenced by the main program. This example program shows more Fortran 95/90 features, including an interface block and an assumed-shape array:

```
! File: array_calc.f90.
! Module containing various calculations on arrays.
MODULE ARRAY_CALCULATOR
INTERFACE
FUNCTION CALC_AVERAGE(D)
REAL :: CALC_AVERAGE
REAL, INTENT(IN) :: D(:)
END FUNCTION CALC_AVERAGE
END INTERFACE
! Other subprogram interfaces...
END MODULE ARRAY_CALCULATOR
```

The example below shows the function declaration CALC_AVERAGE referenced by the main program:

```
! File: calc_aver.f90.
! External function returning average of array.
FUNCTION CALC_AVERAGE(D)
REAL :: CALC_AVERAGE
REAL, INTENT(IN) :: D(:)
CALC_AVERAGE = SUM(D) / UBOUND(D, DIM = 1)
END FUNCTION CALC_AVERAGE
```

Commands to Create a Sample Program

During the early stages of program development, the sample program files shown above might be compiled separately and then linked together, using the following commands:

Linux OS and Mac OS* X example:

1. `ifort -c array_calc.f90`
2. `ifort -c calc_aver.f90`

3. `ifort -c main.f90`
4. `ifort -o calc main.o array_calc.o calc_aver.o`

Windows* example:

1. `ifort /c array_calc.f90`
2. `ifort /c calc_aver.f90`
3. `ifort /c main.f90`
4. `ifort /exe:calc main.obj array_calc.obj calc_aver.obj`

In this sequence of commands:

Line 1: The `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option prevents linking and retains the object files. This command creates the module file `array_calculator.mod` and the object file `array_calc.o` (Linux OS and Mac OS X) or array_calc.obj (Windows OS). Note that the name in the MODULE statement determines the name of module file `array_calculator.mod`. Module files are written into the current working directory.

Line 2: This command creates the object file `calc_aver.o` (Linux OS and Mac OS X) or `calc_aver.obj` (Windows OS).

Line 3: This command creates the object file `main.o` (Linux OS and Mac OS X) or `main.out` (Windows OS)and uses the module file `array_calculator.mod`.

Line 4: This command links all object files into the executable program named `calc`. To link files, use the `ifort` command instead of the `ld` command.

Running the Sample Program

If your path definition includes the directory containing `calc`, you can run the program by simply entering its name:

```
calc
```

When running the sample program, the PRINT and READ statements in the main program result in the following dialogue between user and program:

```
Type five numbers:
55.5
4.5
3.9
9.0
5.6
Average of the five numbers is:   15.70000
```

Debugging the Sample Program

To debug a program with the debugger, compile the source files with the `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) option to request additional symbol table information for source line debugging in the object and executable program files.

The following `ifort` command lines for Linux OS and Mac OS X systems use the `-o` option to name the executable program file `calc_debug`. The Mac OS X command line also uses the `-save-temps` option, which specifies that the object files should be saved; otherwise, they will be deleted by default.

```
ifort -g -o calc_debug array_calc.f90 calc_aver.f90 main.f90 (Linux)
ifort -g -save-temps -o calc_debug array_calc.f90 calc_aver.f90 main.f90 (Mac OS X)
```

The Windows OS equivalent of this command is the following:

```
ifort /debug:full /exe:calc_debug array_calc.f90 calc_aver.f90 main.f90
```

See also Debugging Fortran Programs and related sections.

Redirecting Command-Line Output to Files

When using the command line, you can redirect standard output and standard error to a file. This avoids displaying a lot of text, which will slow down execution; scrolling text in a terminal window on a workstation can cause an I/O bottleneck (increased elapsed time) and use more CPU time.

Linux* OS and Mac OS* X:

The following example applies to Linux* OS and Mac OS* X.

To run the program more efficiently, redirect output to a file and then display the program output:

```
myprog > results.lis
more results.lis
```

Windows* OS:

The following examples apply to Windows OS.

To place standard output into file `one.out` and standard error into file `two.out`, use the `ifort` command as follows:

```
ifort filenames /options 1>one.out 2>two.out
```

You can also use a short-cut form (omit the 1):

```
ifort filenames /options >one.out 2>two.out
```

To place standard output and standard error into a single file `both.out`, enter the `ifort` command as follows:

```
ifort filenames /options 1>both.out 2>&1
```

You can also use a short-cut form (omit the 1):

```
ifort filenames /options >both.out 2>&1
```

Using Makefiles to Compile Your Application

To specify a number of files with various paths and to save this information for multiple compilations, you can use a makefile.

Linux OS and Mac OS X:

To use a makefile to compile your input files, make sure that /usr/bin and /usr/local/bin are in your path.

If you use the C shell, you can edit your `.cshrc` file and add the following:

```
setenv PATH  /usr/bin:/usr/local/bin:yourpath
```

Then you can compile as:

```
make -f yourmakefile
```

where `-f` is the `make` command option to specify a particular makefile.

Windows OS:

To use a makefile to compile your input files, use the `nmake` command. For example, if your project is `your_project.mak`:

```
nmake /f your_project.mak FPP=ifort.exe LINK32=xilink.exe
```

The arguments of this `nmake` command are as follows:

`/f`	A particular makefile.
your_project.mak	A makefile you want to use to generate object and executable files.
FPP	The compiler-invoking command you want to use. The name of this macro might be different for your makefile. This command invokes the preprocessor.
LINK32	The linker you want to use.

The `nmake` command creates object files (`.obj`) and executable files (`.exe`) specified in `your_project.mak` file.

Specifying Memory Models to use with Systems Based on Intel® 64 Architecture

The following applies to Linux* operating systems only.

Applications designed to take advantage of Intel® 64 architecture can be built with one of three memory models:

- `small (-mcmodel=small)`

  This causes code and data to be restricted to the first 2GB of address space so that all accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.

- `medium (-mcmodel=medium)`

  This causes code to be restricted to the first 2GB; however, there is no restriction on data. Code can be addressed with IP-relative addressing, but access of data must use absolute addressing.

- `large (-mcmodel=large)`

  There are no restrictions on code or data; access to both code and data uses absolute addressing.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. This can affect code size and performance. (IP-relative addressing is somewhat faster.)

Additional Notes on Memory Models and on Large Data Objects

- When you specify the medium or large memory models, you must also specify the `-shared-intel` compiler option to ensure that the correct dynamic versions of the Intel run-time libraries are used.
- When you build shared objects (.so), Position-Independent Code (PIC) is specified (that is, `-fpic` is added by the compiler driver) so that a single .so can support all three memory models. However, code that is to be placed in a static library, or linked statically, must be built with the proper memory model specified. Note that there is a performance impact to specifying the medium or large memory models.
- The use of the memory model (medium, large) option and the `-shared-intel` option is required as a by-product of the code models stipulated in the 64-bit Application Binary Interface (ABI), which is written specifically for processors with the 64-bit memory extensions. Both the compiler and the GNU linker (ld) are responsible for generating the proper code and necessary relocations on this platform according to the chosen memory model.

- The 2GB restriction on Intel® 64 architecture involves not only arrays greater than 2GB, but also COMMON blocks and local data with a total size greater than 2GB. The Compiler Options reference contains additional discussion of the supported memory models and offers details about the 2GB restrictions for each model. See the `-mcmodel` options page.

- If, during linking, you fail to use the appropriate memory model and dynamic library options, an error message in this format occurs:

```
<some lib.a library>(some .o): In Function <function>:
  : relocation truncated to fit: R_X86_64_PC32 <some symbol>
```

Allocating Common Blocks

Use the `-dyncom` (Linux OS and Mac OS X) or `/Qdyncom` (Windows OS) option to dynamically allocate common blocks at run time.

This option designates a common block to be dynamic. The space for its data is allocated at run time rather than compile time. On entry to each routine containing a declaration of the dynamic common block, a check is performed to see whether space for the common block has been allocated. If the dynamic common block is not yet allocated, space is allocated at the check time.

The following command-line example specifies the dynamic common option with the names of the common blocks to be allocated dynamically at run time:

```
ifort -dyncom "blk1,blk2,blk3" test.f (Linux OS and Mac OS X)
ifort /Qdyncom"BLK1,BLK2,BLK3" test.f (Windows OS)
```

where `BLK1`, `BLK2`, and `BLK3`are the names of the common blocks to be made dynamic.

Guidelines for Using the dyncom/Qdyncom Option

The following are some limitations that you should be aware of when using the `-dyncom` (Linux OS and Mac OS X) or `/Qdyncom` (Windows OS) option:

- An entity in a dynamic common cannot be initialized in a DATA statement.
- Only named common blocks can be designated as dynamic COMMON.
- An entity in a dynamic common block must not be used in an EQUIVALENCE expression with an entity in a static common block or a DATA-initialized variable.

For more information, see the following topic:

- `-dyncom` compiler option

Why Use a Dynamic Common Block?

A main reason for using dynamic common blocks is to enable you to control the common block allocation by supplying your own allocation routine. To use your own allocation routine, you should link it ahead of the Fortran run-time library. This routine must be written in the C language to generate the correct routine name.

The routine prototype is:

```
void _FTN_ALLOC(void **mem, int *size, char *name);
```

where

- *mem* is the location of the base pointer of the common block which must be set by the routine to point to the block of memory allocated.
- *size* is the integer number of bytes of memory that the compiler has determined are necessary to allocate for the common block as it was declared in the program. You can ignore this value and use whatever value is necessary for your purpose.

NOTE. You must return the size in bytes of the space you allocate. The library routine that calls `_FTN_ALLOC()` ensures that all other occurrences of this common block fit in the space you allocated. Return the size in bytes of the space you allocate by modifying *size*.

- *name* is the name of the common block being dynamically allocated.

Allocating Memory to Dynamic Common Blocks

The run-time library routine, `f90_dyncom`, performs memory allocation. The compiler calls this routine at the beginning of each routine in a program that contains a dynamic common block. In turn, this library routine calls `_FTN_ALLOC()` to allocate memory. By default, the compiler passes the size in bytes of the common block as declared in each routine to `f90_dyncom`, and then on to `_FTN_ALLOC()`. If you use the nonstandard extension having the common block of the same name declared with different sizes in different routines, you might get a run-time error depending on the order in which the routines containing the common block declarations are invoked.

The Fortran run-time library contains a default version of `_FTN_ALLOC()`, which simply allocates the requested number of bytes and returns.

Running Fortran Applications from the Command Line

If you run a program from the command line, the operating system searches directories listed in the PATH environment variable to find the executable file you have requested.

You can also run your program by specifying the complete path of the executable file. On Windows* operating systems, any DLLs you are using must be in the same directory as the executable or in one specified in the path.

Multithreaded Programs

If your program is multithreaded, each thread starts on whichever processor is available at the time. On a computer with one processor, the threads all run in parallel, but not simultaneously; the single processor switches among them. On a computer with more than one processor, the threads can run simultaneously.

Using the fpscomp:filesfromcmd Option

If you specify the `-fpscomp filefromcmd` (Linux OS and Mac OS X) or `/fpscomp:filesfrom-cmd` (Windows OS) option, the command line that executes the program can also include additional filenames to satisfy `OPEN` statements in your program in which the filename field (FILE specifier) has been left blank. The first filename on the command line is used for the first such `OPEN` statement executed, the second filename for the second `OPEN` statement, and so on. (In the Visual Studio IDE, you can provide these filenames using Project>Properties. Choose the Debugging category and enter the filenames in the Command Arguments text box.)

Each filename on the command line (or in an IDE dialog box) must be separated from the names around it by one or more spaces or tab characters. You can enclose each name in quotation marks ("*filename*"), but this is not required unless the argument contains spaces or tabs. A null argument consists of an empty set of quotation marks with no filename enclosed ("").

The following example runs the program MYPROG.EXE from the command line:

```
MYPROG "" OUTPUT.DAT
```

Because the first filename argument is null, the first `OPEN` statement with a blank filename field produces the following message:

```
File name missing or blank - please enter file name
UNIT number  ?
```

The *number* is the unit number specified in the OPEN statement. The filename OUTPUT.DAT is used for the second such OPEN statement executed. If additional OPEN statements with blank filename fields are executed, you will be prompted for more filenames.

Instead of using the -fpscomp filesfromcmd or /fpscomp:filesfromcmd option, you can:

- Call the GETARG library routine to return the specified command-line argument. To execute the program in the Visual Studio IDE, provide the command-line arguments to be passed to the program using Project>Properties. Choose the Debugging category and enter the arguments in the Command Arguments text box .
- On Windows OS, call the GetOpenFileName Windows API routine to request the file name using a dialog box.

For more information, see the following topic:

- fpscomp compiler option

5 Input and Output Files

Understanding Input File Extensions

The Intel® Fortran compiler interprets the type of each input file by the file name extension.

The file extension determines whether a file gets passed to the compiler or to the linker. The following types of files are used with the ifort command:

- Files passed to the compiler: `.f90`, `.for`, `.f`, `.fpp`, `.i`, `.i90`, `.ftn`

  Typical Fortran source files have a file extension of .f90, .for, and .f. When editing your source files, you need to choose the source form, either free-source form or fixed-source form (or a variant of fixed form called tab form). You can either use a compiler option to specify the source form used by the source files (see the description for the free or fixed compiler option) or you can use specific file extensions when creating or renaming your files. For example:

  - The compiler assumes that files with an extension of .f90 or .i90 are free-form source files.
  - The compiler assumes that files with an extension of .f, .for, .ftn, or .i are fixed-form (or tab-form) files.

- Files passed to the linker: `.a`, `.lib`, `.obj`, `.o`, `.exe`, `.res`, `.rbj`, `.def`, `.dll`

The most common file extensions and their interpretations are:

Filename	Interpretation	Action
filename`.a` (Linux* OS and Mac OS* X) *filename*`.lib` (Windows* OS)	Object library	Passed to the linker.
filename`.f` *filename*`.for` *filename*`.ftn` *filename*`.i`	Fortran fixed-form source	Compiled by the Intel® Fortran compiler.

Filename	Interpretation	Action
filename.fpp and, on Linux, filenames with the following uppercase extensions: .FPP, .F, .FOR, .FTN	Fortran fixed-form source	Automatically preprocessed by the Intel Fortran preprocessor fpp; then compiled by the Intel Fortran compiler.
filename.f90 *filename*.i90	Fortran free-form source	Compiled by the Intel Fortran compiler.
filename.F90 (Linux OS and Mac OS X)	Fortran free-form source	Automatically preprocessed by the Intel Fortran preprocessor fpp; then compiled by the Intel Fortran compiler.
filename.s (Linux OS and Mac OS X) *filename*.asm (Windows)	Assembly file	Passed to the assembler.
filename.o (Linux OS and Mac OS X) *filename*.obj (Windows OS)	Compiled object file	Passed to the linker.

When you compile from the command line, you can use the compiler configuration file to specify default directories for input libraries. To specify additional directories for input files, temporary files, libraries, and for the files used by the assembler and the linker, use compiler options that specify output file and directory names.

Producing Output Files

The output produced by the ifort command includes:

- An object file, if you specify the -c (Linux OS and Mac OS X) or /c (Windows OS) option on the command line. An object file is created for each source file.
- An executable file, if you omit the -c or /c option.
- One or more module files (such as datadef.mod), if the source file contains one or more MODULE statements.

- A shareable library (such as `mylib.so` on Linux OS, `mylib.dylib` on Mac OS X, or`mylib.dll` on Windows OS), if you use the `-shared` (Linux),`-dynamiclib` (Mac OS X) or `/libs:dll` (Windows OS) option.
- Assembly files, if you use the `-S` (Linux OS and Mac OS X) or `/S` (Windows OS) option. This creates an assembly file for each source file.

You control the production of output files by specifying the appropriate compiler options on the command line or using the appropriate properties in the integrated development environment for Windows OS and Mac OS X.

For instance, if you do not specify the `-c` or `/c` option, the compiler generates a temporary object file for each source file. It then invokes the linker to link the object files into one executable program file and causes the temporary object files to be deleted.

If you specify the `-c` or `/c` option, object files are created and retained in the current working directory. You must link the object files later. You can do this by using a separate ifort command; alternatively, you can call the linker (`ld` for Linux OS and Mac OS X or `link` for Windows OS) directly to link in objects. On Linux OS and Mac OS X systems, you can also call `xild` or use the archiver (ar) and xiar to create a library. For Mac OS X, you would use `libtool` to generate a library.

If fatal errors are encountered during compilation, or if you specify certain options such as `-c` or `/c`, linking does not occur.

The output files include the following:

Output File	Extension	How Created on the Command Line
Object file	.o (Linux OS and Mac OS X) .obj (Windows OS)	Created automatically.
Executable file	.out (Linux OS and Mac OS X) .exe (Windows OS)	Do not specify `-c` or `/c`.
Shareable library file	.so (Linux OS) .dylib (Mac OS X) .dll (Windows OS)	Specify `-shared` (Linux OS),`-dynamiclib` (Mac OS X) or `/libs:dll` (Windows OS) and do not specify `-c` or `/c` .

Output File	Extension	How Created on the Command Line
Module file	.mod	Created if a source file being compiled defines a Fortran module (MODULE statement).
Assembly file	.s (Linux OS and Mac OS* X) .asm (Windows OS)	Created if you specify the `S` option. An assembly file for each source file is created.

To allow optimization across all objects in the program, use the `-ipo/Qipo` option.

To specify a file name for the executable program file (other than the default) use the `-o` *`output`* (Linux OS and Mac OS X) or `/exe:`*`output`* (Windows OS) option, where `output` specifies the file name.

NOTE. You cannot use the `c` and `o` options together with multiple source files.

Temporary Files Created by the Compiler or Linker

Temporary files created by the compiler or linker reside in the directory used by the operating system to store temporary files.

To store temporary files, the driver first checks for the TMP environment variable. If defined, the directory that TMP points to is used to store temporary files.

If the TMP environment variable is not defined, the driver then checks for the TMPDIR environment variable. If defined, the directory that TMPDIR points to is used to store temporary files.

If the TMPDIR environment variable is not defined, the driver then checks for the TEMP environment variable. If defined, the directory that TEMP points to is used to store temporary files.

For Windows* OS, if the TEMP environment variable is not defined, the current working directory is used to store temporary files. For Linux* OS and Mac OS* X, if the TEMP environment variable is not defined, the `/tmp` directory is used to store temporary files.

Temporary files are usually deleted. Use the `-save-temps` (Linux OS and Mac OS X) or `/Qsave-temps` (Windows OS) compiler option to save temporary files created by the compiler in the current working directory. This option only saves intermediate files that are normally created during compilation.

For performance reasons, use a local drive (rather than a network drive) to contain temporary files.

To view the file name and directory where each temporary file is created, use the `-watch all` (Linux OS and Mac OS X) or `/watch:all` (Windows OS) option.

To create object files in your current working directory, use the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) option.

Any object files that you specify on the command line are retained.

Setting Environment Variables 6

Using the ifortvars File to Specify Location of Components

Before you first invoke the compiler, you need to be sure certain environment variables are set. These environment variables define the location of the various compiler-related components.

The Intel Fortran Compiler installation includes a file that you can run to set environment variables.

- On Linux* OS and Mac OS* X, the file is a shell script called `ifortvars.sh` or `ifortvars.csh`.
- On Windows* OS, the file is a batch file called `ifortvars.bat`.

The following information is operating system-dependent.

Linux OS and Mac OS X:

Set the environment variables before using the compiler. You can use the `source` command to execute the shell script, `ifortvars.sh` or `ifortvars.csh`, from the command line to set them.

The script takes an architecture argument:

- ia32: Compiler and libraries for IA-32 architecture only
- intel64: Compiler and libraries for Intel® 64 architecture only
- ia64: Compiler and libraries for IA-64 architectures only (Linux OS)

For example, to execute this script file for the bash shell:

```
source /opt/intel/Compiler/version_number/package_id/bin/ifortvars.sh ia32
```

If you use the C shell, use the `.csh` version of this script file:

```
source /opt/intel/Compiler/version_number/package_id/bin/ifortvars.csh ia32
```

If you want `ifortvars.sh` to run automatically when you start Linux OS or Mac OS X, you can edit your `.bash_profile` file and add the line above to the end of your file. For example:

```
# set up environment for Intel compiler
source /opt/intel/fc/version_number/package_id/bin/ifortvars.sh ia32
```

If you compile a program without ensuring the proper environment variables are set, you will see an error similar to the following when you execute the compiled program:

```
./a.out: error while loading shared libraries:
libimf.so: cannot open shared object file: No such file or directory
```

Windows OS:

Under normal circumstances, you do not need to run the ifortvars.bat batch file. The Fortran command-line window sets these variables for you automatically.

To activate this command-line window, select **Fortran Build Environment for applications...** available from the **Start>All Programs>Intel(R) Software Development Tools>Intel(R) Visual Fortran Compiler Professional**

NOTE. You will need to run the batch file if you open a command-line window without using the provided Build Environment for applications... menu item in the Intel Fortran program folder or if you want to use the compiler from a script of your own.

The batch file inserts the directories used by Intel Fortran at the beginning of the existing paths. Because these directories appear first, they are searched before any directories in the path lists provided by Windows OS. This is especially important if the existing path includes directories with files having the same names as those needed by Intel Fortran.

If needed, you can run `ifortvars.bat`each time you begin a session on Windows* systems by specifying it as the initialization file with the PIF Editor.

The batch file takes two arguments:

```
<install-dir>\bin\ifortvars.bat <arg1> [<arg2>]
```

`<arg1>` is one of the following

- ia32: Compiler and libraries for IA-32 architecture only
- ia32_intel64: Compiler running on IA-32 architecture that generates code for Intel® 64 architecture; uses Intel® 64 architecture libraries
- ia32_ia64: Compiler running on IA-32 architecture that generates code for IA-64 architecture; uses IA-64 architecture libraries
- intel64: Compiler and libraries for Intel® 64 architecture only
- ia64: Compiler and libraries for IA-64 architectures only

`<arg2>`, if specified, is one of the following:

- vs2005: Microsoft Visual Studio 2005
- vs2008: Microsoft Visual Studio 2008

If <arg2> is not specified, the script uses the version of Visual Studio that was detected during the installation procedure.

Setting Compile-Time Environment Variables

The following table shows the compile-time environment variables that affect the Intel® Fortran Compiler:

Environment Variable	Description
IFORTCFG	Specifies a configuration file that the compiler should use instead of the default configuration file. By default, the compiler uses the default configuration file (ifort.cfg) from the same directory where the compiler executable resides. **NOTE.** On Windows* operating systems, this environment variable cannot be set from the IDE.
INTEL_LICENSE_FILE	Specifies the location of the product license file.
PATH	Specifies the directory path for the compiler executable files.
TMP, TMPDIR, TEMP	Specifies the directory in which to store temporary files. See Temporary Files Created by the Compiler or Linker. **NOTE.** On Windows operating systems, this environment variable cannot be set from the IDE.
FPATH (Linux* OS and Mac OS* X)	The path for include and module files.

Environment Variable	Description
GCCROOT (Linux OS and Mac OS X)	Specifies the location of the gcc binaries. Set this variable only when the compiler cannot locate the gcc binaries when using the -gcc-name option.
GXX_INCLUDE (Linux OS and Mac OS X)	The location of the gcc headers. Set this variable to specify the locations of the GCC installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc.
GXX_ROOT (Linux OS and Mac OS X)	The location of the gcc binaries. Set this variable to specify the locations of the GCC installed files when the compiler does not find the needed values as specified by the use of -gcc-name=*directory-name*/gcc.
LIBRARY_PATH (Linux OS and Mac OS X)	The path for libraries to be used during the link phase.
LD_LIBRARY_PATH (Linux OS)	The path for shared (.so) library files.
DYLD_LIBRARY_PATH (Mac OS X)	The path for dynamic libraries.
INCLUDE (Windows OS)	Specifies the directory path for the include files (files included by an INCLUDE statement, #include files, RC INCLUDE files, and module files referenced by a USE statement).
LIB (Windows OS)	Specifies the directory path for .LIB (library) files, which the linker links in. If the LIB environment variable is not set, the linker looks for .LIB files in the current directory.

Additionally, there are a number of run-time environment variables that you can set. For a list of environment variables recognized at run time and information on setting and viewing environment variables, see Setting Run-Time Environment Variables.

You can use the SET command at the command prompt to set environment variables. Depending on your operating system, there are additional ways to set environment variables.

Setting Environment Variables (Linux OS and Mac OS X)

You can set environment variables by using the `ifortvars.csh` and `ifortvars.sh` files to set several at a time. The files are found in the product's `bin` directory. See Using the ifortvars File to Specify Location of Components.

Within the C Shell, use the setenv command to set an environment variable:

```
setenv FORT9 /usr/users/smith/test.dat
```

To remove the association of an environment variable and its value within the C shell, use the `unsetenv` command.

```
unsetenv FORT9
```

Within the Bourne* shell (sh), the Korn shell (ksh), and the bash shell, use the export command and assignment command to set the environment variable:

```
export FORT9
FORT9=/usr/users/smith/test.dat
```

To remove the association of an environment variable and its value within the Bourne* shell, the Korn shell, or the bash shell, use the unset command:

```
unset FORT9
```

Setting Environment Variables (Windows OS)

Certain environment variables specifying path, library, and include directories can be defined in the IDE on a per user basis using Tools>Options... from the menu bar.

Additionally, you can set the environment variables needed by Intel Fortran using the `ifortvars.bat` file. See Using the ifortvars File to Specify Location of Components.

NOTE. If you specify `devenv/useenv` on the command line to start the IDE, the IDE uses the PATH, INCLUDE, and LIB environment variables defined for that command line when performing a build. It uses these values instead of the values defined using Tool>Options.

For more information on the `devenv` command, see the `devenv` description in the Microsoft Visual Studio* documentation.

During installation, the Intel Fortran compiler may modify certain system-wide environment variables, depending on your installation choices. (For more information, see the install.htm file.)

To view or change these environment variable settings, do the following:

On Windows* 2000, Windows NT* 4, or Windows XP* operating systems:

1. Log into an account with Administrator privilege.
2. Open the Control panel.
3. Click System.
4. On Windows 2000 and Windows XP systems: Click the Advanced tab and then click the Environment Variables button. On Windows NT 4 systems: Click the Environment tab.
5. View or change the displayed environment variables.
6. To have any environment variable changes take effect immediately, click Apply.
7. Click OK.

NOTE. Changing system-wide environment variables affects command line builds (those done without IDE involvement), but not builds done through the IDE. IDE builds are managed by the environment variables set in the IDE Using Tools>Options. An exception to this is an IDE build (`devenv`) done from the command line that specifies the `/useenv` option. In this case, the IDE uses the PATH, INCLUDE, and LIB environment variables defined for that command line.

You can set an environment variable from within a program by calling the SETENVQQ routine. For example:

```
USE IFPORT
 LOGICAL(4) success
 success = SETENVQQ("PATH=c:\mydir\tmp")
 success = &
 SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")
```

Setting Run-Time Environment Variables

The Intel® Fortran run-time system recognizes a number of environment variables. These variables can be used to customize run-time diagnostic error reporting, allow program continuation under certain conditions, disable the display of certain dialog boxes under certain conditions, and allow just-in-time debugging. For a list of run-time environment variables used by OpenMP*, see OpenMP Environment Variables in Optimizing Applications.

For information on setting compile time environment variables, see Setting Compile-Time Environment Variables.

The run-time environment variables are:

- `decfort_dump_flag`

If this variable is set to Y or y, a core dump will be taken when any severe Intel Fortran run-time error occurs.

- `F_UFMTENDIAN`

  This variable specifies the numbers of the units to be used for little-endian-to-big-endian conversion purposes. See Environment `Variable F_UFMTENDIAN` Method.

- `FOR_FMT_TERMINATOR`

  This variable specifies the numbers of the units to have a specific record terminator. See Record Types.

- `FOR_ACCEPT`

  The ACCEPT statement does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the `FOR_ACCEPT` environment variable. If `FOR_ACCEPT` is *`not`* defined, the code `ACCEPT f,iolist` reads from `CONIN$` (standard input). If `FOR_ACCEPT` is defined (as a file name optionally containing a path), the specified file would be read.

- `FOR_DEBUGGER_IS_PRESENT`

  This variable tells the Fortran run-time library that your program is executing under a debugger. If set to True, it generates debug exceptions whenever severe or continuous errors are detected. Under normal conditions you don't need to set this variable on Windows systems, as this information can be extracted from the operating system. On Linux* OS and Mac OS* X, you will need to set this variable if you want debug exceptions. Setting this variable to True when your program is *not* executing under a debugger will cause unpredictable behavior.

- `FOR_DEFAULT_PRINT_DEVICE` (Windows* OS only)

  This variable lets you specify the print device other than the default print device PRN (LPT1) for files closed (`CLOSE` statement) with the `DISPOSE='PRINT'` specifier. To specify a different print device for the file associated with the `CLOSE` statement `DISPOSE='PRINT'` specifier, set `FOR_DEFAULT_PRINT_DEVICE` to any legal DOS print device before executing the program.

- `FOR_DIAGNOSTIC_LOG_FILE`

  If this variable is set to the name of a file, diagnostic output is written to the specified file.

  The Fortran run-time system attempts to open that file (append output) and write the error information (ASCII text) to the file.

  The setting of `FOR_DIAGNOSTIC_LOG_FILE` is independent of `FOR_DISABLE_DIAGNOSTIC_DISPLAY`, so you can disable the screen display of information but still capture the error information in a file. The text string you assign for the file name is used literally, so you must specify the full name. If the file open fails, no error is reported and the run-time system continues diagnostic processing.

See also Locating Run-Time Errors and Using Traceback Information.

- FOR_DISABLE_DIAGNOSTIC_DISPLAY

  This variable disables the display of all error information. This variable is helpful if you just want to test the error status of your program and do not want the Fortran run-time system to display any information about an abnormal program termination.

  See also Using Traceback Information.

- FOR_DISABLE_STACK_TRACE

  This variable disables the call stack trace information that follows the displayed severe error message text.

  The Fortran run-time error message is displayed whether or not FOR_DISABLE_STACK_TRACE is set to true. If the program is executing under a debugger, the automatic output of the stack trace information by the Fortran library will be disabled to reduce noise. You should use the debugger's stack trace facility if you want to view the stack trace.

  See also Locating Run-Time Errors and Using Traceback Information.

- FOR_IGNORE_EXCEPTIONS

  This variable disables the default run-time exception handling, for example, to allow just-in-time debugging. The run-time system exception handler returns EXCEPTION_CONTINUE_SEARCH to the operating system, which looks for other handlers to service the exception.

- FOR_NOERROR_DIALOGS

  This variable disables the display of dialog boxes when certain exceptions or errors occur. This is useful when running many test programs in batch mode to prevent a failure from stopping execution of the entire test stream.

- FOR_PRINT

  Neither the PRINT statement nor a WRITE statement with an asterisk (*) in place of a unit number includes an explicit logical unit number. Instead, both use an implicit internal logical unit number and the FOR_PRINT environment variable. If FOR_PRINT is *not* defined, the code PRINT f,iolist or WRITE (*,f) iolist writes to CONOUT$ (standard output). If FOR_PRINT is defined (as a file name optionally containing a path), the specified file would be written to.

- FOR_READ

  A READ statement that uses an asterisk (*) in place of a unit number does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_READ environment variable. If FOR_READ is *not* defined, the code READ (*,f) iolist or READ f,iolist reads from CONIN$ (standard input). If FOR_READ is defined (as a file name optionally containing a path), the specified file would be read.

- FOR_TYPE

  The TYPE statement does not include an explicit logical unit number. Instead, it uses an implicit internal logical unit number and the FOR_TYPE environment variable. If FOR_TYPE is *not* defined, the code TYPE f,iolist writes to CONOUT$ (standard output). If FOR_TYPE is defined (as a file name optionally containing a path), the specified file would be written to.

- FORT_BUFFERED

  Lets you request that buffered I/O should be used at run time for output of all Fortran I/O units, except those with output to the terminal. This provides a run-time mechanism to support the -assume buffered_io (Linux OS and Mac OS X) or /assume:buffered_io (Windows OS) compiler option.

- FORT_CONVERT*n*

  Lets you specify the data format for an unformatted file associated with a particular unit number (*n*), as described in Methods of Specifying the Data Format.

- FORT_CONVERT.*ext* and FORT_CONVERT_*ext*

  Lets you specify the data format for unformatted files with a particular file extension suffix (ext), as described in Methods of Specifying the Data Format.

- FORT_FMT_RECL

  Lets you specify the default record length (normally 132 bytes) for formatted files.

- FORT_UFMT_RECL

  Lets you specify the default record length (normally 2040 bytes) for unformatted files.

- FORT*n*

  Lets you specify the file name for a particular unit number *n*, when a file name is not specified in the OPEN statement or an implicit OPEN is used, and the compiler option -fpscomp filesfromcmd (Linux OS and Mac OS X) or /fpscomp:filesfromcmd (Windows OS) was not specified. Preconnected files attached to units 0, 5, and 6 are by default associated with system standard I/O files.

- NLSPATH (Linux OS and Mac OS X only)

  The path for the Intel Fortran run-time error message catalog.

- TBK_ENABLE_VERBOSE_STACK_TRACE

  This variable displays more detailed call stack information in the event of an error.

  The default brief output is usually sufficient to determine where an error occurred. Brief output includes up to twenty stack frames, reported one line per stack frame. For each frame, the image name containing the PC, routine name, line number, and source file are given.

The verbose output, if selected, will provide (in addition to the information in brief output) the exception context record if the error was a machine exception (machine register dump), and for each frame, the return address, frame pointer and stack pointer and possible parameters to the routine. This output can be quite long (but limited to 16K bytes) and use of the environment variable `FOR_DIAGNOSTIC_LOG_FILE` is recommended if you want to capture the output accurately. Most situations should not require the use of verbose output.

The variable `FOR_ENABLE_VERBOSE_STACK_TRACE` is also recognized for compatibility with Compaq* Visual Fortran.

See also Using Traceback Information.

- `TBK_FULL_SRC_FILE_SPEC`

  By default, the traceback output displays only the file name and extension in the source file field. To display complete file name information including the path, set the environment variable `TBK_FULL_SRC_FILE_SPEC` to true.

  The variable `FOR_FULL_SRC_FILE_SPEC` is also recognized for compatibility with Compaq* Visual Fortran.

  See also Using Traceback Information.

- `FORT_TMPDIR`, `TMP`, `TMPDIR`, and `TEMP`

  Specifies an alternate working directory where scratch files are created.

Setting Environment Variables within a Program

You can set a run-time environment variable from within a program by calling the SETENVQQ routine. For example:

```
program ENVVAR
use ifport
LOGICAL(4) res
! Add other data declarations here
! call SETENVQQ as a function
res=SETENVQQ("FOR_IGNORE_EXCEPTIONS=T")
```

Using Compiler Options

7

Compiler Options Overview

A compiler option (also known as a switch) is an optional string of one or more alphanumeric characters preceded by a dash (-) (Linux* OS and Mac OS* X) or a forward slash (/) (Windows*OS).

Some options are on by default when you invoke the compiler.

Depending on your operating system, compiler options are typically specified in the following ways:

- On the compiler command line
- In the IDE, either Xcode (Mac OS X) or Microsoft Visual Studio* (Windows OS)

Most compiler options perform their work at compile time, although a few apply to the generation of extra code used at run time.

For more information about compiler options, see the Compiler Options reference.

For information on the option mapping tool, which shows equivalent options between Windows and Linux OS, see the Option Mapping Tool.

Getting Help on Options

For help, enter `-help [category]` (Linux OS and Mac OS X) or `/help [category]` (Windows) on the command line, which displays brief information about all the command-line options or a specific category of compiler options.

The Compiler Options reference provides a complete description of each compiler option, including the `-help` option.

NOTE. If there are enabling and disabling versions of options on the command line, or two versions of the same option, the last one takes precedence.

Using Multiple ifort Commands

If you compile parts of your program by using multiple ifort commands, options that affect the execution of the program should be used consistently for all compilations, especially if data is shared or passed between procedures. For example:

- The same data alignment needs to be used for data passed or shared by module definitions (such as user-defined structures) or common blocks. Use the same version of the `-align` (Linux OS and Mac OS X) or `/align` (Windows) option for all compilations.
- The program might contain INTEGER, LOGICAL, REAL, or COMPLEX declarations without a kind parameter or size specifier that is passed or shared by module definitions or common blocks. You must consistently use the options that control the size of such numeric data declarations.

Using the OPTIONS Statement to Override Options

You can override some options specified on the command line by using the OPTIONS statement in your Fortran source program. The options specified by the OPTIONS statement affect only the program unit where the statement occurs.

Using the Option Mapping Tool

The Intel compiler's Option Mapping Tool provides an easy method to derive equivalent options between Windows* and Linux*operating systems. If you are a Windows OS developer who is developing an application for Linux OS, you may want to know, for example, the Linux OS equivalent for the `/Oy-` option. Likewise, the Option Mapping Tool provides Windows OS equivalents for Intel compiler options supported on Linux OS.

NOTE. The Compiler Option Mapping Tool does not run on Mac OS* X.

Using the Compiler Option Mapping Tool

You can start the Option Mapping Tool from the command line by:

- invoking the compiler and using the `-map-opts` option
- or, executing the tool directly

NOTE. The Compiler Option Mapping Tool only maps compiler options on the same architecture. It will not, for example, map an option that is specific to the IA-64 architecture to a like option available on the IA-32 architecture or Intel® 64 architecture.

Calling the Option Mapping Tool with the Compiler

If you use the compiler to execute the Option Mapping Tool, the following syntax applies:

```
<compiler command> <map-opts option> <compiler option(s)>
```

Example: Finding the Windows OS equivalent for `-fp`

```
ifort -map-opts -fp
Intel(R) Compiler option mapping tool
mapping Linux OS options to Windows OS for Fortran
'-map-opts' Linux OS option maps to
  --> '-Qmap-opts' option on Windows OS
  --> '-Qmap_opts' option on Windows OS
'-fp' Linux OS option maps to
  --> '-Oy-' option on Windows OS
```

NOTE. Output from the Option Mapping Tool also includes:

- option mapping information (not shown here) for options included in the compiler configuration file
- alternate forms of the options that are supported but may not be documented

Calling the Option Mapping Tool Directly

Use the following syntax to execute the Option Mapping Tool directly from a command line environment where the full path to the `map-opts` executable is known (compiler `bin` directory):

```
map-opts -t<target OS> -l<language> -opts <compiler option(s)>
```

where values for:

- `<target OS> = {l|linux|w|windows}`
- `<language> = {f|fortran|c}`

Example: Finding the Windows equivalent for `-fp`

```
map-opts -tw -lf -opts -fp
Intel(R) Compiler option mapping tool
mapping Linux OS options to Windows OS for Fortran
'-fp' Linux OS option maps to
  --> '-Oy-' option on Windows OS
```

Compiler Directives Related to Options

Some compiler directives and compiler options have the same effect, as shown in the table below. However, compiler directives can be turned on and off throughout a program, while compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

Compiler directives and equivalent command-line compiler options are:

Compiler Directive	Equivalent Command-Line Compiler Option
DECLARE	-warn declarations (Linux* OS and Mac OS* X) /warn:declarations or /4Yd (Windows* OS)
NODECLARE	-warn nodeclarations (Linux OS and Mac OS X) /warn:nodeclarations or /4Nd (Windows OS)
DEFINE *symbol*	-D*name* (Linux OS and Mac OS X) /define:symbol or /D*name* (Windows OS)
FIXEDFORMLINESIZE:*option*	-extend-source [option] (Linux OS and Mac OS X) /extend-source[:*n*] or /4L*n* (Windows OS)
FREEFORM	-free or -nofixed (Linux OS and Mac OS X) /free or /nofixed or /4Yf (Windows OS)
NOFREEFORM	-nofree or -fixed (Linux OS and Mac OS X) /nofree or /fixed or /4Nf (Windows OS)
INTEGER:*option*	-integer_size option (Linux OS and Mac OS X) /integer_size:*option* or /4I*option* (Windows OS)
OBJCOMMENT	/libdir:user (Windows OS)
OPTIMIZE [:*n*]	-O (Linux OS and Mac OS X) or /O (Windows OS) *n* is 0, 1, 2, or 3 for opt levels -O0 through -O3. If *n* is omitted, default is 2.

Compiler Directive	Equivalent Command-Line Compiler Option
NOOPTIMIZE	-O0 (Linux OS and Mac OS X) or /Od (Windows OS)
PACK:*option*	-align [*option*] (Linux OS and Mac OS X) /align[:*n*] or /Zp*n* (Windows OS)
REAL:*option*	-real-size*option* (Linux OS and Mac OS X) /real-size: *option* or /4R*option* (Windows)
STRICT	-warn stderrors with -stand (Linux OS and Mac OS X) /warn:stderrors with /stand:f90 or /4Ys (Windows)
NOSTRICT	-warn nostderrors (Linux OS and Mac OS X) /warn:nostderrors or /4Ns (Windows OS)

NOTE. For Windows OS, the compiler directive names above are specified using the prefix !DEC$ followed by a space; for example: !DEC$ NOSTRICT. The prefix !DEC$ works for both fixed-form and free-form source. You can also use these alternative prefixes for fixed-form source only: cDEC$, CDEC$, *DEC$, cDIR$, CDIR$, *DIR$, and !MS$.

For more information on compiler directives, see Directive Enhanced Compilation.

8 Preprocessing

Using the fpp Preprocessor

If you choose to preprocess your source programs, you can use the preprocessor fpp, which is the preprocessor supplied with the Intel® Fortran Compiler, or the preprocessing directives capability of the Fortran compiler. It is recommended that you use fpp.

The Fortran preprocessor, fpp, is provided as part of the Intel® Fortran product. When you use a preprocessor for Intel Fortran source files, the generated output files are used as input source files by the Compiler.

Preprocessing performs such tasks as preprocessor symbol (macro) substitution, conditional compilation, and file inclusion. Intel Fortran predefined symbols are described in Predefined Preprocessor Symbols.

The Compiler Options reference provides syntactical information on fpp. Additionally, it contains a list of fpp options that are available when fpp is in effect.

Automatic Preprocessing by the Compiler

By default, the preprocessor is not run on files before compilation. However, the Intel Fortran compiler automatically calls fpp when compiling source files that have a filename extension of .fpp, and, on Linux* OS and Mac OS* X, file extensions of .F, .F90, .FOR, .FTN, or .FPP. For example, the following command preprocesses a source file that contains fpp preprocessor directives, then passes the preprocessed file to the compiler and linker:

```
ifort source.fpp
```

If you want to preprocess files that have other Fortran extensions than those listed, you have to explicitly specify the preprocessor with the `-fpp` compiler option.

The fpp preprocessor can process both free- and fixed-form Fortran source files. By default, filenames with the suffix of .F, .f, .for, or .fpp are assumed to be in fixed form. Filenames with a suffix of .F90 or .f90 (or any other suffix not specifically mentioned here) are assumed to be free form. You can use the `-free` (Linux OS and Mac OS X) or /free (Windows* OS) option to specify free form and the `-fixed` (Linux OS and Mac OS X) or `/fixed` (Windows OS) option to explicitly specify fixed form.

The fpp preprocessor recognizes tab format in a source line in fixed form.

Running fpp to Preprocess Files

You can explicitly run fpp in these ways:

- On the `ifort` command line, use the `ifort` command with the `-fpp` (Linux OS and Mac OS X) or `/fpp` (Windows OS) option. By default, the specified files are then compiled and linked. To retain the intermediate (.i or .i90) file, specify the `-save-temps` (Linux OS and Mac OS X) or `/Qsave-temps` (Windows OS) option.
- On the command line, use the fpp command. In this case, the compiler is not invoked. When using the fpp command line, you need to specify the input file and the intermediate (.i or .i90) output file. For more information, type `fpp -help` (Linux OS and Mac OS X) or fpp `/help` (Windows OS) on the command line.
- In the Microsoft Visual Studio* IDE, set the Preprocess Source File option to Yes in the Fortran Preprocessor Option Category. To retain the intermediate files, add `/Qsave-temps` to Additional Options in the Fortran Command Line Category.

fpp has some of the capabilities of the ANSI C preprocessor and supports a similar set of directives. Directives must begin in column 1 of any Fortran source files.Preprocessor directives are not part of the Fortran language and not subject to the rules for Fortran statements. Syntax for directives is based on that of the C preprocessor.

The following lists some common cpp features that are supported by fpp; it also shows common cpp features that are not supported.

Supported cpp features:	**Unsupported cpp features:**
#define, #undef, #ifdef, #ifndef, #if, #elif, #else, #endif, #include, #error, #warning, #line	#pragma, #ident
# (stringsize) and ## (concatenation) operators	spaces or tab characters preceding the initial "#" character
	# followed by empty line
! as negation operator	\ backslash-newline

Unlike cpp, fpp does not merge continued lines into a single line when possible.

You do not usually need to specify preprocessing for Fortran source programs unless your program uses fpp preprocessing commands, such as those listed above.

CAUTION. Using a preprocessor that does not support Fortran can damage your Fortran code, especially with `FORMAT (\\I4)` with cpp changes the meaning of the program because the `double` backslash "`\\`" indicates end-of-record with most C/C++ preprocessors.

fpp Source Files

A source file can contain fpp tokens in the form of :

- fpp directive names. For more information on directives, see Using fpp Directives.
- symbolic names including Fortran keywords. fpp permits the same characters in names as Fortran. For more information on symbolic names, see Using Predefined Preprocessor Symbols.
- constants. Integer, real, double, and quadruple precision real, binary, octal, hexadecimal (including alternate notation), character, and Hollerith constants are allowed.
- special characters, space, tab and newline characters
- comments, including:
  - Fortran language comments. A fixed form source line containing one of the symbols C, c, *, d, or D in the first position is considered a comment line. The ! symbol is interpreted as the beginning of a comment extending to the end of the line except when the "!" occurs within a constant-expression in a #if or #elif directive. Within such comments, macro expansions are not performed, but they can be switched on by `-f-com=no`.
  - fpp comments between / and */. They are excluded from the output and macro expansions are not performed within these symbols. fpp comments can be nested: for each /* there must be a corresponding */. fpp comments are useful for excluding from the compilation large portions of source instead of commenting every line with a Fortran comment symbol.
  - C++ -like line comments which begin with // (double-slash).

A string that is a token can occupy several lines, but only if its input includes continued line characters using the Fortran continuation character &. fpp will not merge such lines into one line.

Identifiers are always placed on one line by fpp. For example, if an input identifier occupies several lines, it will be merged by fpp into one line.

fpp Output

Output consists of a modified copy of the input, plus lines of the form:

```
#line_number  file_name
```

These are inserted to indicate the original source line number and filename of the output line that follows. Use the fpp option `-P` (Linux OS and Mac OS X) or `/P` (Windows OS) to disable the generation of these lines.

Diagnostics

There are three kinds of fpp diagnostic messages:

- warnings: preprocessing of source code is continued and the fpp return value is 0
- errors: fpp continues preprocessing but sets the return value a nonzero value which is the number of errors
- fatal errors: fpp stops preprocessing and returns a nonzero return value.

The messages produced by fpp are intended to be self-explanatory. The line number and filename where the error occurred are displayed along with the diagnostic on stderr.

Using fpp Directives

All fpp directives start with the number sign (#) as the first character on a line. White space (blank or tab characters) can appear after the initial "#" for indentation.

fpp directives (beginning with the # symbol in the first position of lines) can be placed anywhere in a source code, in particular before a Fortran continuation line. However, fpp directives within a macro call may not be divided among several lines by means of continuation symbols.

fpp directives can be grouped according to their purpose.

Directives for string substitution

fpp contains directives that result in substitutions in a user's program:

Directive	Result
`__FILE__`	replace this string with the input file name (a character string literal)
`__LINE__`	replace this string with the current line number in the input file (an integer constant)

Directive	Result
__DATE__	replace this string with the date that fpp processed the input file (a character string literal in the form `Mmm dd yyyy`)
__TIME__	replace this string with the time that fpp processed the input file (a character string literal in the form `hh:mm:ss`)

Directive for inclusion of external files

There are two forms of file inclusion:

`#include "filename"`

`#include <filename>`

This directive reads in the contents of the named file into this location in the source. The lines read in from the file are processed by fpp just as if they were part of the current file.

When the <*filename*> notation is used, *filename* is only searched for in the standard "include" directories. See the `-I` option and also the `-Y` option for more detail. No additional tokens are allowed on the directive line after the final '"' or ">".

Files are searched for in the following order:

- for `#include "filename"`:
  - in the directory in which the source file resides
  - in the directories specified by the `-I` or `-Y` option
  - in the default directory
- for `#include <filename>`:
  - in the directories specified by the `-I` or `-Y` option
  - in the default directory

Directive for line control

This directive takes the following form:

`#line-number "filename"`

This directive generates line control information for the Fortran compiler. *line-number* is an integer constant that is the line number of the next line. "*filename*" is the name of the file containing the line. If "*filename*" is not given, the current filename is assumed.

Directive for fpp variable and macro definitions

The #define directive, used to define both simple string variables and more complicated macros, takes the two forms.

The first form is the definition of an fpp variable:

```
#define name token-string
```

In the above, occurrences of *name* in the source file will be substituted with *token-string*.

The second form is the definition of an fpp macro.

```
#define name(argument[,argument] ... ) token-string
```

In the above, occurrences of the macro *name* followed by the comma-separated list of actual arguments within parentheses are replaced by token-string with each occurrence of each *argument* in token-string replaced by the token sequence representing the corresponding "actual" argument in the macro call.

An error is produced if the number of macro call arguments is not the same as the number of arguments in the corresponding macro definition. For example, consider this macro definition:

```
#define INTSUB(m, n, o) call mysub(m, n, o)
```

Any use of the macro INTSUB must have three arguments. In macro definitions, spaces between the macro name and the open parenthesis "(" are prohibited to prevent the directive from being interpreted as an fpp variable definition with the rest of the line beginning with the open parenthesis "(" being interpreted as its token-string.

An fpp variable or macro definition can be of any length and is limited only by the newline symbol. It can be defined in multiple lines by continuing it to the next line with the insertion of "\". The occurrence of a newline without a macro-continuation signifies the end of the macro definition.

Example:

```
#define long_macro_name(x,\
   y) x*y
```

The scope of a definition begins from the #define and encloses all the source lines (and source lines from #include files) to the end of the current file, except for:

- files included by Fortran INCLUDE statements

- fpp and Fortran comments
- Fortran IMPLICIT statements that specify a single letter
- Fortran FORMAT statements
- numeric, typeless, and character constants

Directive for undefining a macro

This directive takes the following form:

```
#undef name
```

This directive removes any definition for *name* (produced by -D options, #define directives, or by default). No additional tokens are permitted on the directive line after *name*.

If *name* has not been defined earlier, then the #undef directive has no effect.

Directive for macro expansion

If, during expansion of a macro, the column width of a line exceeds column 72 (for fixed format) or column 132 (for free format), fpp inserts appropriate Fortran continuation lines.

For fixed format, there is a limit on macro expansions in label fields (positions 1-5):

- a macro call (together with possible arguments) should not extend beyond column 5
- a macro call whose name begins with one of the Fortran comment symbols is considered to be part of a comment
- a macro expansion may produce text extending beyond column 5. In this case, a warning will be issued

In fixed format, when the fpp `-Xw` option has been specified, an ambiguity may occur if a macro call occurs in a statement position and a macro name begins or coincides with a Fortran keyword. For example, consider the following:

```
#define callp(x)   call f(x)
  call p(0)
```

fpp cannot determine how to interpret the "call p" token sequence above. It could be considered as a macro name. The current implementation does the following:

- the longer identifier is chosen (callp in this case)
- from this identifier the longest macro name or keyword is extracted

- if a macro name has been extracted a macro expansion is performed. If the name begins with some keyword, fpp issues an appropriate warning
- the rest of the identifier is considered as a whole identifier

In the previous example, the macro expansion is performed and the following warning is produced:

```
warning: possibly incorrect substitution of macro callp
```

This situation appears only when preprocessing a fixed format source code and when the space symbol is not interpreted as a token delimiter.

In the following case, a macro name coincides with a beginning of a keyword:

```
 #define INT  INTEGER*8
              INTEGER k
```

The INTEGER keyword will be found earlier than the INT macro name. There will be no warning when preprocessing such a macro definition.

Directives for conditional selection of source text

There are three forms of conditional selection of source text.

Form 1:

```
#if condition_1
                block_1
            #elif condition_2
                block_2
            #elif ...
            #else
                 block_n
            #endif
```

Form 2:

```
          #ifdefname
                block_1
            #elif condition
                block_2
            #elif ...
            #else
                block_n
            #endif
```

Form 3:

```
#ifndef name
        block_1
   #elif condition
        block_2
   #elif ...
   #else
        block_n
   #endif
```

The `elif` and `else` parts are optional in all three forms. There may be more than one `elif` part in each form.

Conditional expressions

condition_1, *condition_2*, etc. are logical expressions involving fpp constants, macros, and intrinsic functions. The following items are permitted:

- C language operations: <, >, ==, !=, >=, <=, +, -, /, *, %, <<, >>, &, ~, |, &&, || They are interpreted by fpp in accordance to the C language semantics (this facility is provided for compatibility with "old" Fortran programs using cpp)
- Fortran language operations: .AND., .OR., .NEQV., .XOR., .EQV., .NOT., .GT., .LT., .LE., .GE., .NE., .EQ., ** (power).
- Fortran logical constants: .TRUE. , .FALSE.
- the fpp intrinsic function "`defined`": `defined(name)` or `defined name` which returns .TRUE. if name is defined as an fpp variable or a macro or returns .FALSE. if the name is not defined

`#ifdef` is a shorthand for `#if defined(name)` and `#ifndef` is a shorthand for `#if .not. defined(name)`.

Only these items, integer constants, and names can be used within a constant-expression. A name that has not been defined with the `-D` option, a `#define` directive, or by default, has a value of 0. The C operation `!=` (not equal) can be used in `#if` or `#elif` directive, but not in the `#define` directive, where the symbol `!` is considered as the Fortran comment symbol by default.

Conditional constructs

The following table summarizes conditional constructs.

Construct	Result
#if *condition*	Subsequent lines up to the matching #else, #elif, or #endif directive appear in the output only if *condition* evaluates to .TRUE. .
#ifdef *name*	Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if name has been defined, either by a #define directive or by the -D option, with no intervening #undef directive. No additional tokens are permitted on the directive line after *name*.
#ifndef *name*	Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if *name* has not been defined, or if its definition has been removed with an #undef directive. No additional tokens are permitted on the directive line after *name*.
#elif *condition*	Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if all of the following occur: • The condition in the preceding #if directive evaluates to .FALSE. or the name in the preceding #ifdef directive is not defined, or the name in the preceding #ifndef directive is defined. • The conditions in all of the preceding #elif directives evaluate to .FALSE. • The condition in the current #elif evaluates to .TRUE. Any condition allowed in an #if directive is allowed in an #elif directive. Any number of #elif directives may appear between an #if, #ifdef, or #ifndef directive and a matching #else or #endif directive.
#else	Subsequent lines up to the matching #endif appear in the output only if all of the following occur: • The condition in the preceding #if directive evaluates to .FALSE. or the name in the preceding #ifdef directive is not defined, or the name in the preceding #ifndef directive is defined. • The conditions in all of the preceding #elif directives evaluate to .FALSE.
#endif	End a section of lines begun by one of the conditional directives #if, #ifdef, or #ifndef. Each such directive must have a matching #endif.

Using Predefined Preprocessor Symbols

Preprocessor symbols (macros) let you substitute values in a program before it is compiled. The substitution is performed in the preprocessing phase.

Some preprocessor symbols are predefined by the compiler system and are available to compiler directives and to fpp. If you want to use others, you need to specify them on the command line.

You can use the `-D` (Linux* OS and Mac OS* X) or `/D` (Windows* OS) option to define the symbol names to be used during preprocessing. This option performs the same function as the `#define` preprocessor directive.

For more information, see the following topic:

- D compiler option

Preprocessing with fpp replaces every occurrence of the defined symbol name with the specified value. Preprocessing compiler directives only allow IF and IF DEFINED.

If you want to disable symbol replacement (also known as macro expansion) during the preprocessor step, you can specify the following: `-fpp:"-macro=no"` (Linux OS and Mac OS X or `/fpp:"/macro=no"` (Windows OS).

Disabling preprocessor symbol replacement is useful for running fpp to perform conditional compilation (using #ifdef, etc.) without replacement.

You can use the `-U` (Linux OS and Mac OS X) or `/U` (Windows OS) option to suppress an automatic definition of a preprocessor symbol. This option suppresses any symbol definition currently in effect for the specified *`name`*. This option performs the same function as an #undef preprocessor directive.

For more information, see the following topic:

- U compiler option

Windows OS:

The following information applies to Windows operating systems.

In addition to specifying preprocessor symbols on the command line, you can also specify them in the Visual Studio* integrated development environment (IDE). To do this, select Project>Properties and use the Preprocessor Definitions item in the General Options or Preprocessor Options category.

The following preprocessor symbols are available:

Predefined Symbol Name and Value	Conditions When this Symbol is Defined
__INTEL_COMPILER=1110	Identifies the Intel Fortran compiler
__INTEL_COMPILER_BUILD_DATE=*YYYYMMDD*	Identifies the Intel Fortran compiler build date
_DLL=1	Only if `/libs:dll`, `/MDs`, `/MD`, `/dll`, or `/LD` is specified, but *not* when `/libs:static` is specified
_MT=1	Only if `/threads` or `/MT` is specified
_M_IX86=*n*00	Only for systems based on IA-32 architecture; *n* is the number specified for `/G` (for example, `_M_IX86=700` for `/G7`)
_M_IA64=64*n*00	Only for systems based on IA-64 architecture; *n* is the number specified for `/G` (for example, `_M_IA64=64200` for `/G1`)
_M_X64	Only for systems based on Intel® 64 architecture. For use in conditionalizing applications for the Intel® 64 platform.
_M_AMD64	Only for systems based on Intel® 64 architecture. This symbol is set by default.
_OPENMP=200805	Valid when OpenMP processing has been requested (that is, when `/Qopenmp` is specified) Takes the form *YYYYMM* where *YYYY* is the year and *MM* is the month of the OpenMP Fortran specification supported. This symbol can be used in both fpp and the Fortran compiler conditional compilations.
_PGO_INSTRUMENT	Defined if `/Qprof_gen` is specified
_WIN32	Always defined
_WIN64	Only for systems based on Intel® 64 architecture and systems based on IA-64 architecture
_VF_VER=1110	Valid when Compaq* Visual Fortran-compatible compile commands (df or f90) are used

When using the non-native IA-64 architecture based compiler, platform-specific symbols are set for the target platform of the executable, not for the system in use.

Linux OS and Mac OS X:

The following information applies to Linux OS and Mac OS X systems.

Symbol Name	Default	Architecture (IA-32, Intel® 64, IA-64)	Description
__INTEL_COMPILER=*n*	On, *n*=1110	All	Identifies the Intel Fortran Compiler
__INTEL_COMPILER_BUILD_DATE =*YYYYMMDD*		All	Identifies the Intel Fortran Compiler build date
__linux__ (Linux only) __linux (Linux only) __gnu_linux__ (Linux only) linux (Linux only) __unix__ (Linux only) __unix (Linux only) unix (Linux only) __ELF__ (Linux only)		All	Defined at the start of compilation
__APPLE__ (Mac OS X only) __MACH__ (Mac OS X only)		IA-32	Defined at the start of compilation
__i386__ __i386 i386		IA-32	Identifies the architecture for the target hardware for which programs are being compiled

Symbol Name	Default	Architecture (IA-32, Intel® 64, IA-64)	Description
__ia64__ (Linux only) __ia64 (Linux only)		IA-64	Identifies the architecture for the target hardware for which programs are being compiled
__x86_64 __x86_64__		Intel® 64	Identifies the architecture for the target hardware for which programs are being compiled.
_OPENMP=*n*	*n*=200805	All	Takes the form *YYYYMM*, where *YYYY* is the year and *MM* is the month of the OpenMP Fortran specification supported. This preprocessor symbol can be used in both fpp and the Fortran compiler conditional compilation. It is available only when -openmp is specified.
_PGO_INSTRUMENT	Off	All	Defined when -prof-gen is specified.
__PIC__ __pic__	Off (Linux OS), On (Mac OS X)	All	Set if the code was requested to be compiled as position independent code. On Mac OS X, these symbols are always set.

Using Configuration Files and Response Files

9

Configuration Files and Response Files Overview

Configuration files and response files let you enter command-line options in a file. Both types of files provide the following benefits:

- Decrease the time you spend entering command-line options
- Ensure consistency of often used commands

See these topics:

- Using Configuration Files
- Using Response Files

Using Configuration Files

Configuration files are automatically processed every time you run the compiler. You can insert any valid command-line options into the configuration file. The compiler processes options in the configuration file in the order in which they appear, followed by the command-line options that you specify when you invoke the compiler.

NOTE. Options in the configuration file are executed every time you run the compiler. If you have varying option requirements for different projects, use response files.

By default, a configuration file named `ifort.cfg` is used. This file resides in the same directory where the compiler executable resides. However, if you want the compiler to use another configuration file in a different location, you can use the `IFORTCFG` environment variable to assign the directory and file name for the configuration file.

Sample Configuration Files

Examples that follow illustrate sample configuration files. The pound (`#`) character indicates that the rest of the line is a comment.

Linux* OS and Mac OS* X Example:

```
## Example ifort.cfg file
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Set extended-length source lines.
-extend_source
##
## Set maximum floating-point significand precision.
-pc80
##
```

Windows* OS Example:

```
#
# Sample ifort.cfg file
## Define preprocessor macro MY_PROJECT
/DMY_PROJECT

## Set extended-length source lines.
/extend_source
##
## Set maximum floating-point significand precision.
/Qpc80
##

## Additional directories to be searched for include
## files, before the default.

/Ic:\project\include


## Use the static, multithreaded run-time library.

/MT
```

Using Response Files

You can use response files to:

- Specify options used during particular compilations for particular projects
- Save this information in individual files

Unlike configuration files, which are automatically processed every time you run the compiler, response files must be invoked as an option on the command line. If you create a response file but do not specify it on the command line, it will not be invoked.

Options specified in a response file are inserted in the command line at the point where the response file is invoked.

You can place any number of options or filenames on a line in the response file. Several response files can be referenced in the same command line.

The syntax for using response files is:

```
ifort @responsefile1 [@responsefile2 ... ]
```

NOTE. An "at" sign (@) must precede the name of the response file on the command line.

Debugging 10

Debugging Fortran Programs

Depending on your operating system and your architecture platform, several debuggers may be available to you.

You can use the debugger provided by your operating system. On Linux* OS and Mac OS* X, this debugger is gdb. On Windows* OS, the debugger is the Microsoft integrated debugger.

On Linux OS and Mac OS X systems, you can also use the Intel® Debugger to debug Intel® Fortran programs.

For more information on IDB, see the Intel Debugger online documentation.

NOTE. On Linux OS and Mac OS X systems, use of the IDB debugger is recommended.

Preparing Your Program for Debugging

This section describes preparing your program for debugging.

Preparing for Debugging using the Command Line

To prepare your program for debugging when using the command line (ifort command):

1. Correct any compilation and linker errors.
2. In a command window, (such as the Fortran command window available from the Intel Fortran program folder), compile and link the program with full debug information and no optimization:

   `ifort -g file.f90` (Linux OS and Mac OS X)

   `ifort /debug:full file.f90` (Windows OS)

On Linux OS and Mac OS X, specify the `-g` compiler option to create unoptimized code and provide the symbol table and traceback information needed for symbolic debugging. (The `-notraceback` option cancels the traceback information.)

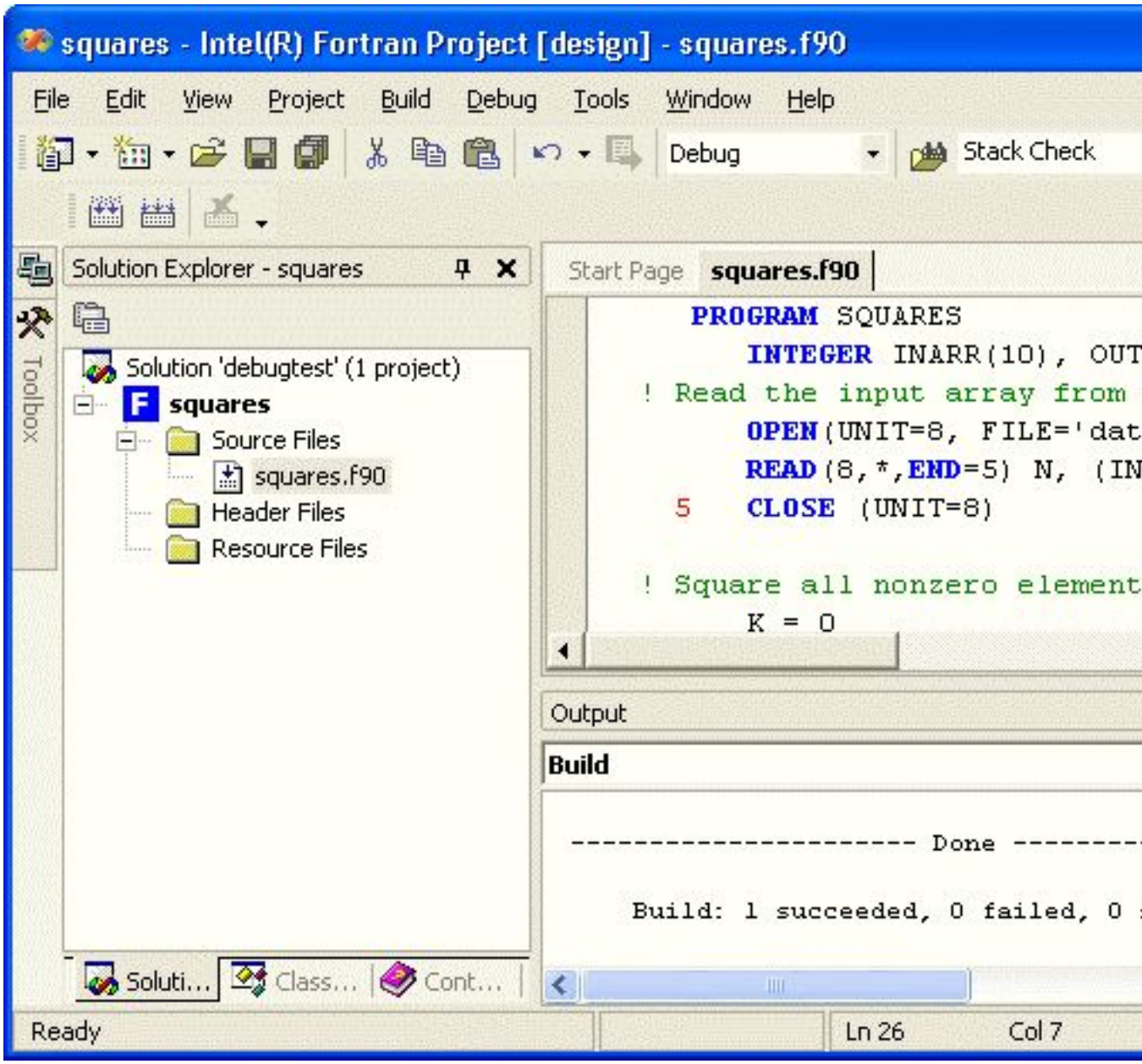
On Windows OS, specify the `/debug:full` compiler option to produce full debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking.

Preparing for Debugging using Microsoft Visual Studio*

The following applies to Windows* operating systems only.

To prepare your program for debugging when using the integrated development environment (IDE):

1. Start the IDE (select the appropriate version of Microsoft Visual Studio in the program folder).
2. Open the appropriate solution (using the Solution menu, either Open Solution or Recent Projects).
3. Open the Solution Explorer View.
4. To view the source file to be debugged, double-click on the file name. The screen resembles the following:


squares - Intel(R) Fortran Project [design] - squares.f90
File Edit View Project Build Debug Tools Window Help
Debug
Stack Check
Solution Explorer - squares
Solution 'debugtest' (1 project)
squares
Source Files
squares.f90
Header Files
Resource Files
Toolbox
Start Page
squares.f90
PROGRAM SQUARES
INTEGER INARR(10), OUT
! Read the input array from
OPEN(UNIT=8, FILE='dat
READ(8,*,END=5) N, (IN
5 CLOSE (UNIT=8)
! Square all nonzero element
K = 0
Output
Build
---------- Done ---------
Build: 1 succeeded, 0 failed, 0
Soluti...
Class...
Cont...
Ready
Ln 26
Col 7

5. In the Build menu, select Configuration Manager and select the Debug configuration.
6. To check your project settings for compiling and linking, select the project name in the Solution Explorer. Now, in the Project menu, select Properties, then click the Fortran folder in the left pane. Similarly, to check the debug options set for your project (such as command arguments or working directory), click the Debugging folder in the Property Pages dialog box.
7. To build your application, select Build>Build Solution.
8. Eliminate any compiler diagnostic messages using the text editor to resolve problems detected in the source code and recompile if needed.
9. Set breakpoints in the source file and debug the program.

Locating Unaligned Data

Unaligned data can slow program execution. You should determine the cause of the unaligned data, fix the source code (if necessary), and recompile and relink the program.

If your program encounters unaligned data at run time, to make it easier to debug the program, you should recompile and relink with the `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) option to generate sufficient table information and debug unoptimized code.

For more information on data alignment, see the following:

Understanding Data Alignment

Setting Data Type and Alignment

Debugging a Program that Encounters a Signal or Exception

If your program encounters a signal (exception) at run time, you may want to recompile and relink with certain command-line options before debugging the cause. The following will make it easier to debug the program:

- Use the `-fpe`*n* (Linux OS and Mac OS X) or `/fpe:n` (Windows OS) option to control the handling of floating point exceptions.
- As with other debugging tasks, use the `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) compiler option to generate sufficient symbol table information and debug unoptimized code.

Debugging an Exception in the Microsoft Debugger

The following applies to Windows* operating systems.

You can request that the program always stop when a certain type of exception occurs. Certain exceptions are caught by default by the Intel Visual Fortran run-time library, so your program stops in the run-time library code. In most cases, you want the program to stop in your program's source code instead .

To change how an exception is handled in the Microsoft debugger:

1. In the Debug menu, select Exceptions.
2. View the displayed exceptions.
3. Select Windows Exceptions. Select each type of exception to be changed and change its handling using the radio buttons.
4. Start program execution using Start in the Debug menu.
5. When the exception occurs, you can now view the source line being executed, examine current variable values, execute the next instruction, and so on to help you better understand that part of your program.
6. After you locate the error and correct the program, consider whether you want to reset the appropriate type of exception to "Use Parent Setting" before you debug the program again.

For machine exceptions, you can use the just-in-time debugging feature to debug your programs as they run outside of the visual development environment. To do this, set the following items:

- In Tools>Options, select Native in the Debugging Just-In Time category.
- Set the `FOR_IGNORE_EXCEPTIONS` environment variable to `TRUE`.

Debugging and Optimizations

This topic describes the relationship between various command-line options that control debugging and optimizing.

Whenever you enable debugging with `-g` (Linux* OS and Mac OS* X) or `/debug:full` (Windows* OS), you disable optimizations. You can override this behavior by explicitly specifying compiler options for optimizations on the command line.

The following summarizes commonly used options for debugging and for optimization.

`-O0` (Linux OS and Mac OS X) or `/Od` (Windows* OS)	Disables optimizations so you can debug your program before any optimization is attempted. This is the default behavior when debugging. On Linux OS and Mac OS X, `-fno-omit-frame-pointer` is set if option -O0 (or -g) is specified. For more information, see the following topic: • `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS) compiler option
`-O1` or `/O1` `-O2` or `/O2 -O3` or `/O3`	Specifies the code optimization level for applications. If you use any of these options, it is recommended that you use `-debug extended` when debugging. For more information, see the following topic: • `-O1`, `-O2`, `-O3` (Linux OS and Mac OS X) or `/O` (Windows OS) compiler option
`-g` or `/debug:full`	Generates symbolic debugging information and line numbers in the object code for use by the source-level debuggers. Turns off `-O2` (Linux OS and Mac OS X) or `/O2` (Windows) and makes `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS)the default. The exception to this is if `-O2`, `-O1` or `-O3` (Linux OS and Mac OS X) or `/O2`, `/O1` or `/O3` (Windows OS) is explicitly specified in the command line. For more information, see the following topic: • `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) compiler option
`-debug extended` (Linux OS and Mac OS X)	Specifies settings that enhance debugging. For more information, see the following topic: • `-debug extended` (Linux OS and Mac OS X)
`-fp` or `/Oy-` (IA-32 architecture only)	Disables the ebp register in optimizations and sets the ebp register to be used as the frame pointer. For more information, see the following topic: • `-fp` (Linux OS and Mac OS X) or `/Oy` (Windows OS) compiler option

`-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)	Causes the compiler to generate extra information in the object file, which allows a symbolic stack traceback. For more information, see the following topic: • `-traceback` compiler option

Combining Optimization and Debugging

The compiler lets you generate code to support symbolic debugging when one of the `O1`, `O2`, or `O3` optimization options is specified on the command line along with `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS); this produces symbolic debug information in the object file.

Note that if you specify an `O1`, `O2`, or `O3` option with the -g or `/debug:full` option, some of the debugging information returned may be inaccurate as a side-effect of optimization. To counter this on Linux OS and Mac OS X, you should also specify the `-debug extended` option.

It is best to make your optimization and/or debugging choices explicit:

- If you need to debug your program excluding any optimization effect, use the `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS) option, which turns off all the optimizations.
- If you need to debug your program with optimizations enabled, then you can specify the `O1`, `O2`, or `O3` option on the command line along with `debug extended`.

NOTE. When no optimization level is specified, the `-g` or `/debug:full` option slows program execution; this is because this option turns on `-O0` or `/Od`, which causes the slowdown. However, if, for example, both `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS) and `-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS) are specified, the code should not experience much of a slowdown.

Refer to the table below for the summary of the effects of using the `-g` or `/debug:full` option with the optimization options.

These options	Produce these results
`-g` (Linux OS and Mac OS X) or `/debug:full` (Windows OS)	Debugging information produced, `-O0` or `/Od` enabled (meaning optimizations are disabled). For Linux OS and Mac OS X, `-fp` is also enabled for compilations targeted for IA-32 architecture.

These options	Produce these results
`-g -O1` (Linux OS and Mac OS X) or `/debug:full /O1` (Windows*)	Debugging information produced, O1 optimizations enabled.
`-g -O2` (Linux OS and Mac OS X) or `/debug:full /O2` (Windows OS)	Debugging information produced, O2 optimizations enabled.
`-g -O2` (Linux OS and Mac OS X) or `/debug:full /O2 /Oy-` (Windows OS)	Debugging information produced, O2 optimizations enabled; for Windows OS using IA-32 architecture, `/Oy` disabled.
`-g -O3 -fp` (Linux OS and Mac OS X) or `/debug:full /O3` (Windows OS)	Debugging information produced, O3 optimizations enabled; for Linux OS, `-fp` enabled for compilations targeted for IA-32 architecture.

NOTE. Even the use of `debug extended` with optimized programs may not allow you to examine all variables or to set breaks at all lines, due to code movement or removal during the optimization process

Debugging Multithreaded Programs

The debugging of multithreaded program discussed in this topic applies to both the OpenMP* Fortran API and the Intel Fortran parallel compiler directives. When a program uses parallel decomposition directives, you must take into consideration that the bug might be caused either by an incorrect program statement or it might be caused by an incorrect parallel decomposition directive. In either case, the program to be debugged can be executed by multiple threads simultaneously.

To determine the correctness of and debug multithreaded programs, you can use the following:

- For Linux OS and Mac OS X systems, Intel® Debugger (IDB) or GDB
- For Windows operating systems, Microsoft Visual Studio* Debugger

- Intel Fortran Compiler debugging options and methods; in particular, `-debug` and `-traceback` (Linux OS and Mac OS X) or `/debug` and `/traceback` (Windows OS)

Data and I/O

11

Data Representation

Data Representation Overview

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte). For information on using nonnative big endian and VAX* floating-point formats, see Supported Native and Nonnative Numeric Formats.

The symbol :A in any figure specifies the address of the byte containing bit 0, which is the starting address of the represented data element.

The following table lists the intrinsic data types used by Intel® Fortran, the storage required, and valid ranges. For information on declaring Fortran intrinsic data types, see Type Declaration Statements. For example, the declaration INTEGER(4) is the same as INTEGER(KIND=4) and INTEGER*4.

Table 21: Fortran Data Types and Storage

Data Type	Storage	Description
BYTE INTEGER(1)	1 byte (8 bits)	A BYTE declaration is a signed integer data type equivalent to INTEGER(1).
INTEGER	See INTEGER(2), INTEGER(4), and INTEGER(8)	Signed integer, either INTEGER(2), INTEGER(4), or INTEGER(8). The size is controlled by the `-integer-size` (Linux* OS and Mac OS* X) or `/integer-size` (Windows* OS) compiler option.
INTEGER(1)	1 byte (8 bits)	Signed integer value from -128 to 127.
INTEGER(2)	2 bytes (16 bits)	Signed integer value from -32,768 to 32,767.
INTEGER(4)	4 bytes (32 bits)	Signed integer value from -2,147,483,648 to 2,147,483,647.

Data Type	Storage	Description
INTEGER(8)	8 bytes (64 bits)	Signed integer value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
REAL	See REAL(4), REAL(8) and REAL(16).	Real floating-point values, either REAL(4), REAL(8), or REAL(16). The size is controlled by the `-real-size` (Linux OS and Mac OS X) or `/real-size` (Windows) compiler option.
REAL(4)	4 bytes (32 bits)	Single-precision real floating-point values in IEEE S_floating format ranging from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are denormalized (subnormal).
REAL(8)	8 bytes (64 bits)	Double-precision real floating-point values in IEEE T_floating format ranging from 2.2250738585072013D-308 to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.9406564584124654D-324 are denormalized (subnormal).
REAL(16)	16 bytes (128 bits)	Extended-precision real floating-point values in IEEE-style X_floating format ranging from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
COMPLEX	See COMPLEX(4), COMPLEX(8) and COMPLEX(16).	Complex floating-point values in a pair of real and imaginary parts that are either REAL(4), REAL(8), or REAL(16). The size is controlled by the `-real-size` (Linux OS and Mac OS X) or `/real-size` (Windows OS) compiler option.
COMPLEX(4)	8 bytes (64 bits)	Single-precision complex floating-point values in a pair of IEEE S_floating format parts: real and imaginary. The real and imaginary parts each range from 1.17549435E-38 to 3.40282347E38. Values between 1.17549429E-38 and 1.40129846E-45 are denormalized (subnormal).
COMPLEX(8) DOUBLE COMPLEX	16 bytes (128 bits)	Double-precision complex floating-point values in a pair of IEEE T_floating format parts: real and imaginary. The real and imaginary parts each range from 2.2250738585072013D-308

Data Type	Storage	Description
		to 1.7976931348623158D308. Values between 2.2250738585072008D-308 and 4.9406564584124654D-324 are denormalized (subnormal).
COMPLEX(16)	32 bytes (256 bits)	Extended-precision complex floating-point values in a pair of IEEE-style X_floating format parts: real and imaginary. The real and imaginary parts each range from 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932.
LOGICAL	See LOGICAL(2), LOGICAL(4), and LOGICAL(8).	Logical value, either LOGICAL(2), LOGICAL(4), or LOGICAL(8). The size is controlled by the `-integer-size` (Linux OS and Mac OS X) or `/integer-size` (Windows OS) compiler option.
LOGICAL(1)	1 byte (8 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(2)	2 bytes (16 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(4)	4 bytes (32 bits)	Logical values .TRUE. or .FALSE.
LOGICAL(8)	8 bytes (64 bits)	Logical values .TRUE. or .FALSE.
CHARACTER	1 byte (8 bits) per character	Character data represented by character code convention. Character declarations can be in the form CHARACTER(LEN=*n*) or CHARACTER**n*, where *n* is the number of bytes or *n* is (*) to indicate passed-length format.
HOLLERITH	1 byte (8 bits) per Hollerith character	Hollerith constants.

In addition, you can define binary (bit) constants.

See these topics:

See Also

- Data Representation
- Integer Data Representations
- Logical Data Representations
- Character Representation
- Hollerith Representation

Integer Data Representations

Integer Data Representations Overview

The Fortran numeric environment is flexible, which helps make Fortran a strong language for intensive numerical calculations. The Fortran standard purposely leaves the precision of numeric quantities and the method of rounding numeric results unspecified. This allows Fortran to operate efficiently for diverse applications on diverse systems.

The effect of math computations on integers is straightforward:

- Integers of KIND=1 consist of a maximum positive integer (127), a minimum negative integer (-128), and all integers between them including zero.
- Integers of KIND=2 consist of a maximum positive integer (32,767), a minimum negative integer (-32,768) , and all integers between them including zero.
- Integers of KIND=4 consist of a maximum positive integer (2,147,483,647), a minimum negative integer (-2,147,483,648), and all integers between them including zero.
- Integers of KIND=8 consist of a maximum positive integer (9,223,372,036,854,775,807), a minimum negative integer (-9,223,372,036,854,775,808), and all integers between them including zero.

Operations on integers usually result in other integers within this range. Integer computations that produce values too large or too small to be represented in the desired KIND result in the loss of precision. One arithmetic rule to remember is that integer division results in truncation (for example, 8/3 evaluates to 2).

Integer data lengths can be 1, 2, 4, or 8 bytes in length.

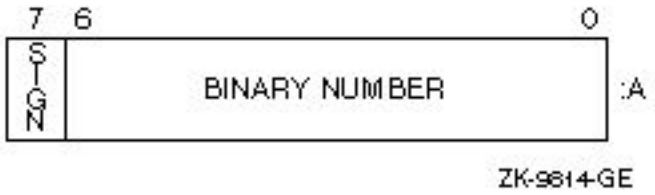
The default data size used for an INTEGER data declaration is INTEGER(4) (same as INTEGER(KIND=4), unless the `-integer-size 16` (Linux* OS and Mac OS* X) or `/integer-size:16` (Windows*OS) or the `-integer-size 64` (Linux OS and Mac OS X) or `/integer-size:64` (Windows OS) option was specified.

Integer data is signed with the sign bit being 0 (zero) for positive numbers and 1 for negative numbers.

INTEGER(KIND=1) Representation

INTEGER(1) values range from -128 to 127 and are stored in 1 byte, as shown below.

Figure 1: INTEGER(1) Data Representation



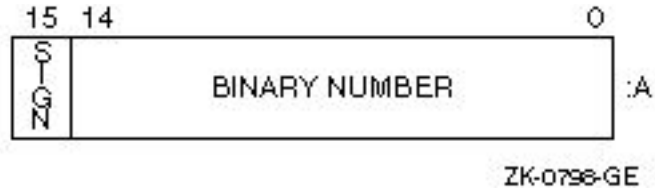
Integers are stored in a two's complement representation. For example:

```
+22 = 16(hex)-7
 = F9(hex)
```

INTEGER(KIND=2) Representation

INTEGER(2) values range from -32,768 to 32,767 and are stored in 2 contiguous bytes, as shown below:

Figure 2: INTEGER(2) Data Representation



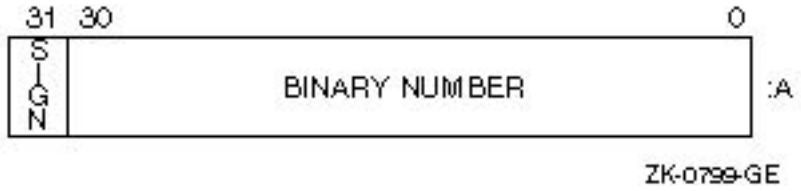
Integers are stored in a two's complement representation. For example:

```
+22 = 0016(hex) -7
 = FFF9(hex)
```

INTEGER(KIND=4) Representation

INTEGER(4) values range from -2,147,483,648 to 2,147,483,647 and are stored in 4 contiguous bytes, as shown below.

Figure 3: INTEGER(4) Data Representation

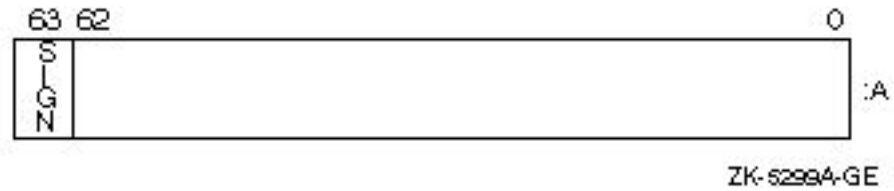


Integers are stored in a two's complement representation.

INTEGER(KIND=8) Representation

INTEGER(8) values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 and are stored in 8 contiguous bytes, as shown below.

Figure 4: INTEGER(8) Data Representation



Integers are stored in a two's complement representation.

Logical Data Representations

Logical data lengths can be 1, 2, 4, or 8 bytes in length.

The default data size used for a LOGICAL data declaration is LOGICAL(4) (same as LOGICAL(KIND=4)), unless `-integer-size 16` or `-integer-size 64` (Linux OS and Mac OS X) or `/integer-size:16` or `/integer-size:64` (Windows OS) was specified.

To improve performance on systems using Intel® 64 architecture and IA-64 architecture, use LOGICAL(4) (or LOGICAL(8)) rather than LOGICAL(2) or LOGICAL(1). On systems using IA-32 architecture, use LOGICAL(4) rather than LOGICAL(8), LOGICAL(2), or LOGICAL(1).

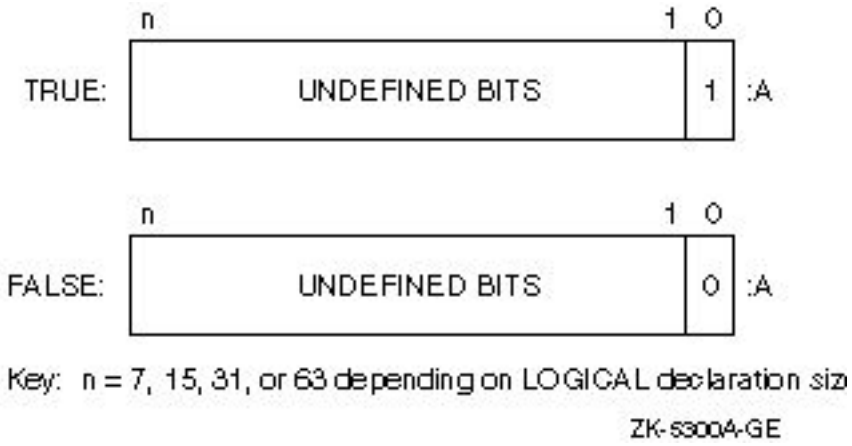
LOGICAL(KIND=1) values are stored in 1 byte. In addition to having logical values .TRUE. and .FALSE., LOGICAL(1) data can also have values in the range -128 to 127. Logical variables can also be interpreted as integer data.

In addition to LOGICAL(1), logical values can also be stored in 2 (LOGICAL(2)), 4 (LOGICAL(4)), or 8 (LOGICAL(8)) contiguous bytes, starting on an arbitrary byte boundary.

If the `-fpscomp nological` (Linux OS and Mac OS X) or `/fpscomp:nological` (Windows OS) compiler option is set (the default), the low-order bit determines whether the logical value is true or false. Specify `logical` instead of `nological` for Microsoft* Fortran PowerStation logical values, where 0 (zero) is false and non-zero values are true.

LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) data representation (when `nological` is specified) appears below.

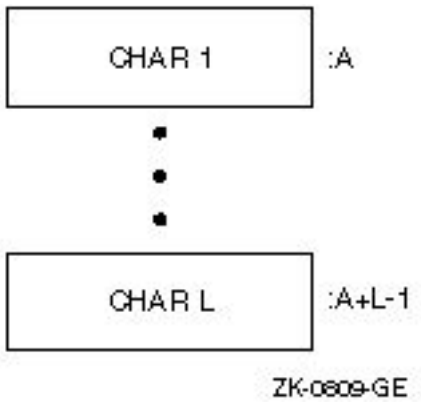
Figure 5: LOGICAL(1), LOGICAL(2), LOGICAL(4), and LOGICAL(8) Data Representations



Character Representation

A character string is a contiguous sequence of bytes in memory, as shown below.

Figure 6: CHARACTER Data Representation

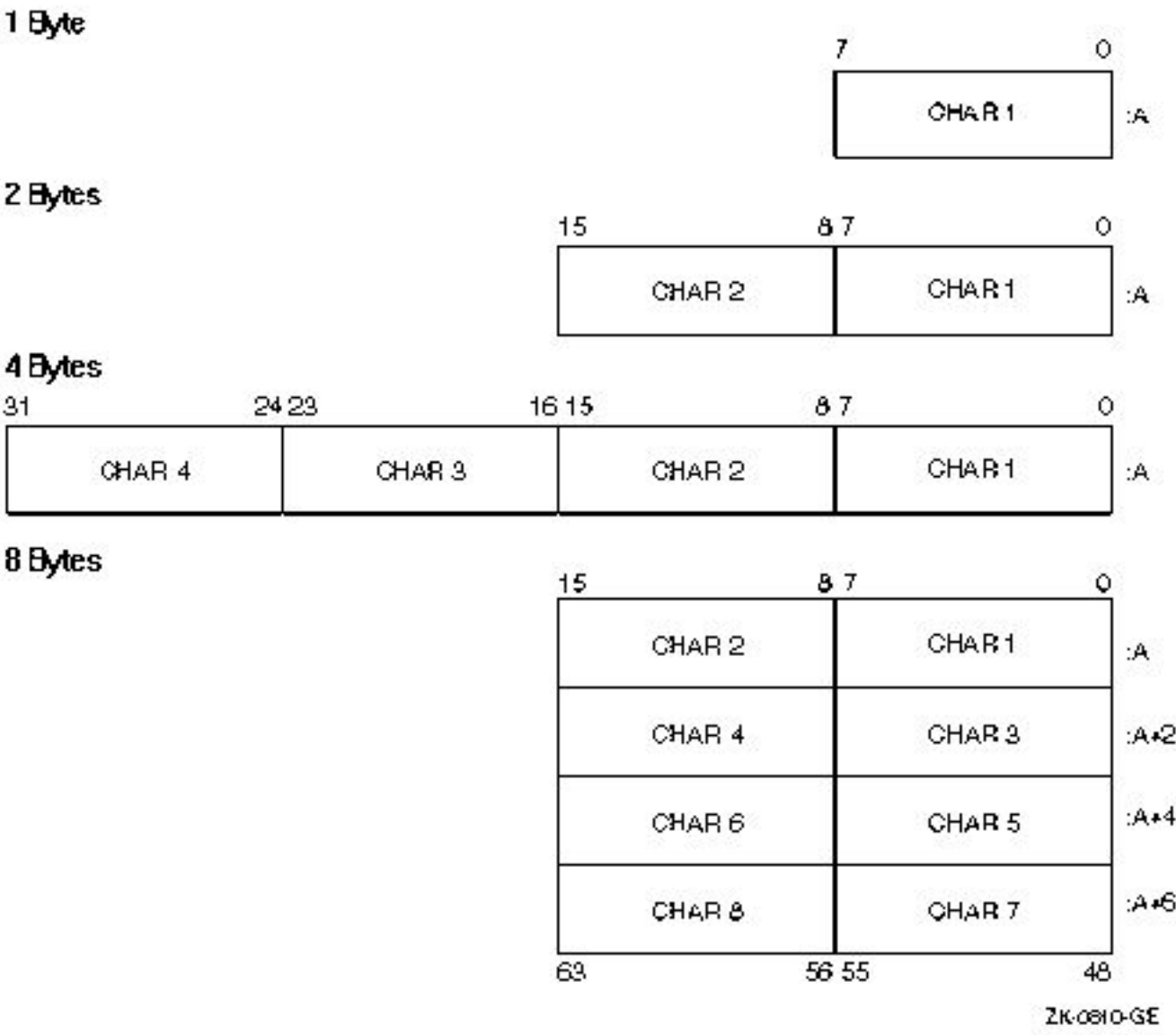


A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. For Windows* OS, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) . For Linux* OS, the length L of a string is in the range 1 through 2,147,483,647 (2**31-1) for systems based on IA-32 architecture and in the range 1 through 9,223,372,036,854,775,807 (2**63-1) for systems based on Intel® 64 architecture and systems based on IA-64 architecture.

Hollerith Representation

Hollerith constants are stored internally, one character per byte, as shown below.

Figure 7: Hollerith Data Representation



Using Traceback Information

Supported Native and Nonnative Numeric Formats

Data storage in different computers uses a convention of either little endian or big endian storage. The storage convention generally applies to numeric values that span multiple bytes, as follows:

Little endian storage occurs when:

- The least significant bit (LSB) value is in the byte with the lowest address.
- The most significant bit (MSB) value is in the byte with the highest address.
- The address of the numeric value is the byte containing the LSB. Subsequent bytes with higher addresses contain more significant bits.

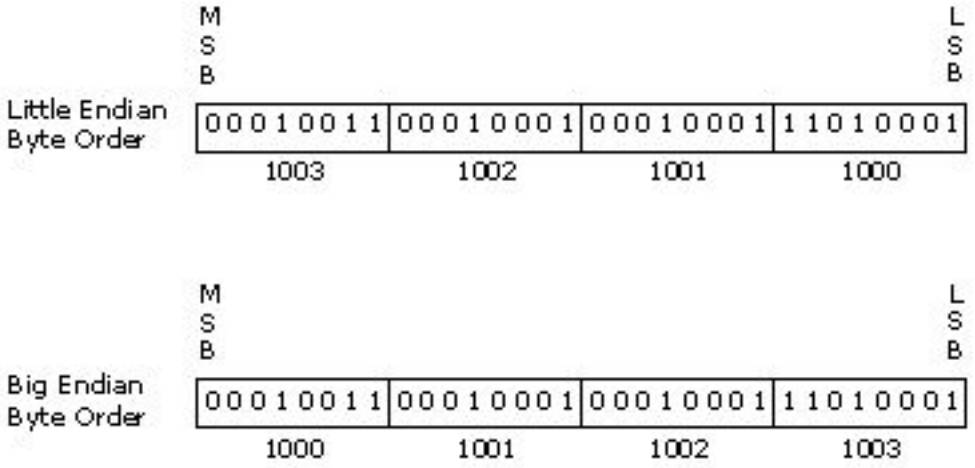
Big endian storage occurs when:

- The least significant bit (LSB) value is in the byte with the highest address.
- The most significant bit (MSB) value is in the byte with the lowest address.
- The address of the numeric value is the byte containing the MSB. Subsequent bytes with higher addresses contain less significant bits.

Intel® Fortran expects numeric data to be in native little endian order, in which the least-significant, right-most zero bit (bit 0) or byte has a lower address than the most-significant, left-most bit (or byte).

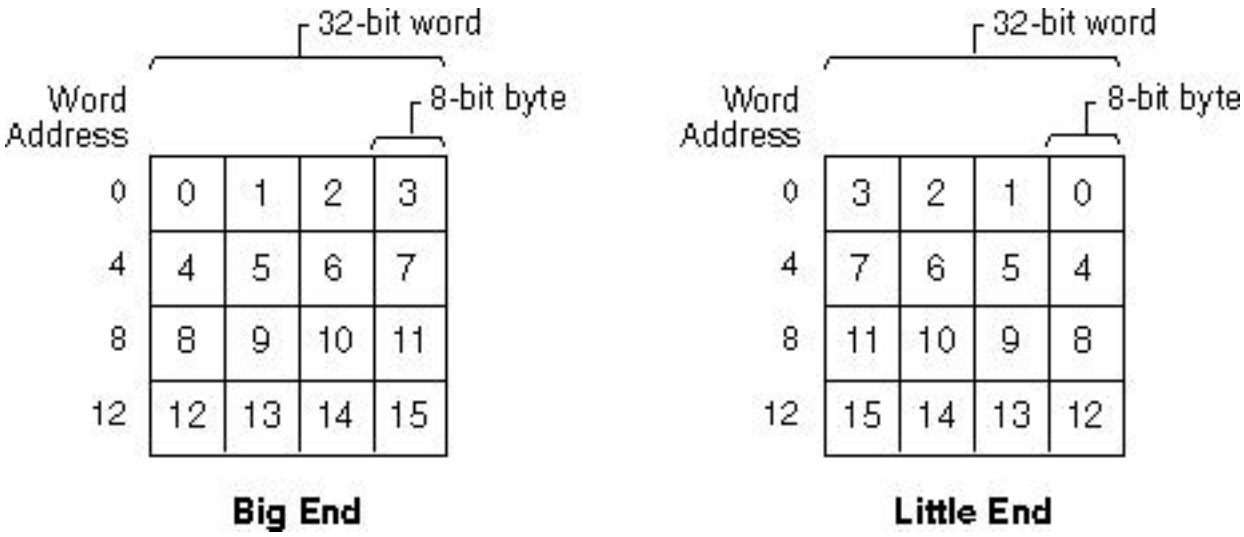
The following figures show the difference between the two byte-ordering schemes:

Figure 8: Little and Big Endian Storage of an INTEGER Value



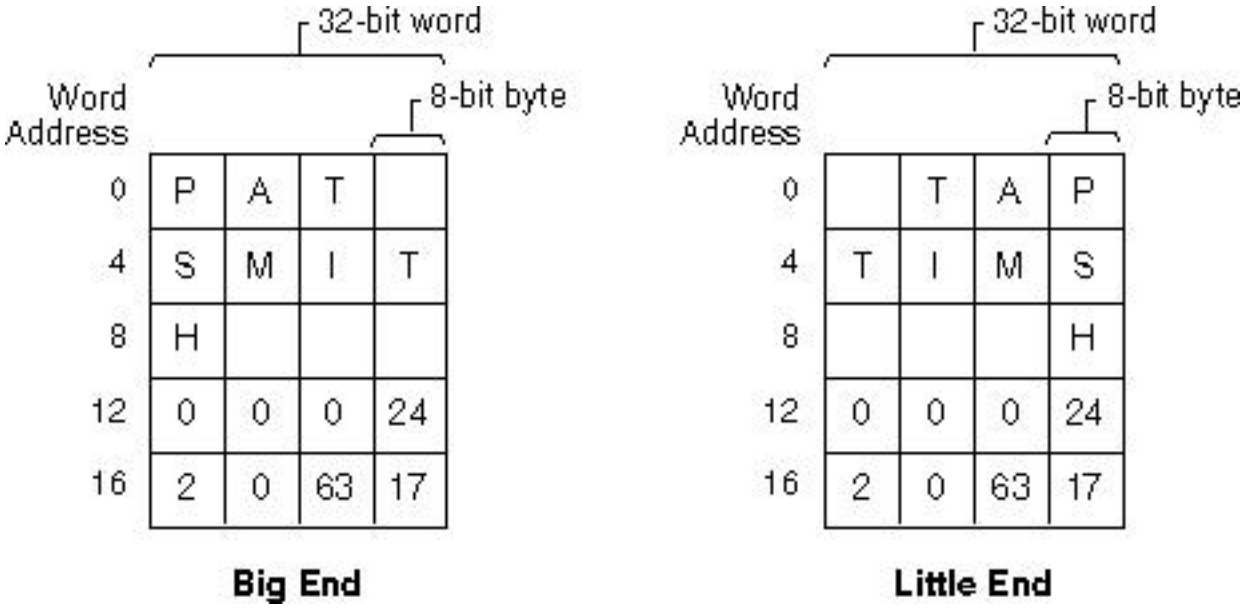
The following figure illustrates the difference between the two conventions for the case of addressing bytes within words.

Figure 9: Byte Order Within Words: (a) Big Endian, (b) Little Endian



Data types stored as subcomponents (bytes stored in words) end up in different locations within corresponding words of the two conventions. The following figure illustrates the difference between the representation of several data types in the two conventions. Letters represent 8-bit character data, while numbers represent the 8-bit partial contribution to 32-bit integer data.

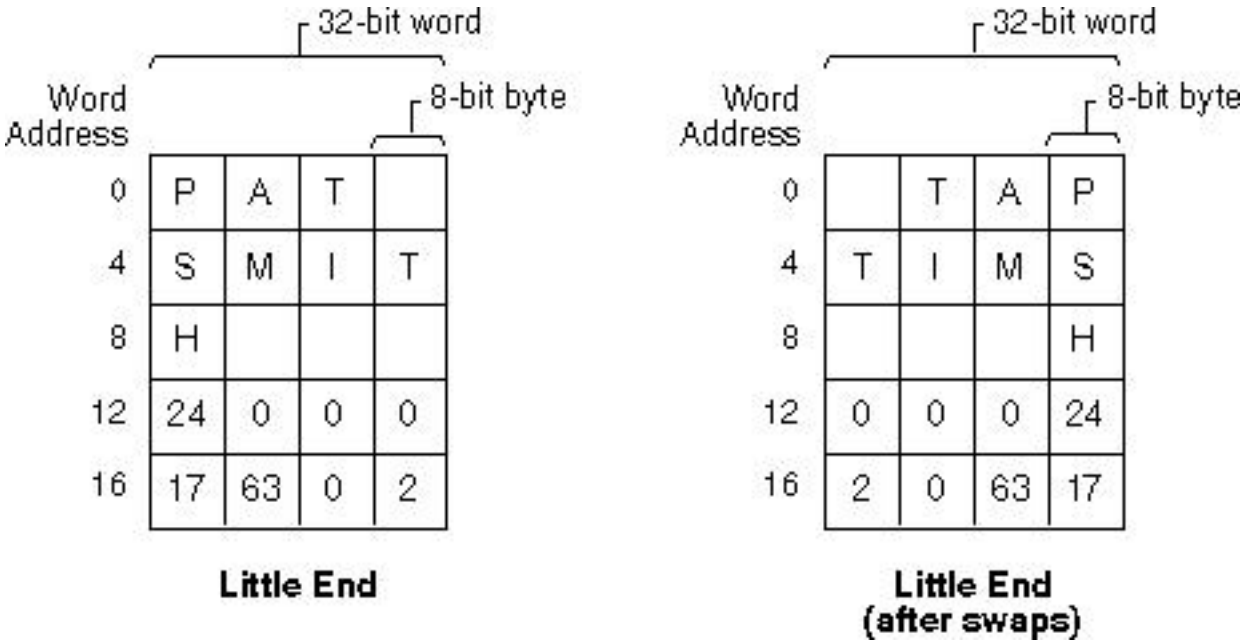
Character and Integer Data in Words: (a) Big Endian, (b) Little Endian



If you serially transfer bytes now from the big endian words to the little endian words (BE byte 0 to LE byte 0, BE byte 1 to LE byte 1, ...), the left half of the figure shows how the data ends up in the little endian words. Note that data of size one byte (characters in this case) is ordered correctly, but that integer data no longer correctly represents the original binary values. The

right half of the figure shows that you need to swap bytes around the middle of the word to reconstitute the correct 32-bit integer values. After swapping bytes, the two preceding figures are identical.

Figure 10: Data Sent from Big to Little: (a) After Transfer, (b) After Byte Swaps



You can generalize the previous example to include floating-point data types and to include multiple-word data types.

Moving unformatted data files between big endian and little endian computers requires that the data be converted.

Intel Fortran provides the capability for programs to read and write unformatted data (originally written using unformatted I/O statements) in several nonnative floating-point formats and in big endian INTEGER or floating-point format. Supported nonnative floating-point formats include Compaq* VAX* little endian floating-point formats supported by Digital* FORTRAN for OpenVMS* VAX Systems, standard IEEE big endian floating-point format found on most Sun Microsystems* systems and IBM RISC* System/6000 systems, IBM floating-point formats (associated with the IBM's System/370 and similar systems), and CRAY* floating-point formats.

Converting unformatted data instead of formatted data is generally faster and is less likely to lose precision of floating-point numbers.

The native memory format includes little endian integers and little endian IEEE floating-point formats, S_floating for REAL(KIND=4) and COMPLEX(KIND=4) declarations, T_floating for REAL(KIND=8) and COMPLEX(KIND=8) declarations, and X_floating for REAL(KIND=16) and COMPLEX(KIND=16) declarations.

The keywords for supported nonnative unformatted file formats and their data types are listed in the following table:

Table 22: Nonnative Numeric Formats, Keywords, and Supported Data Types

Keyword	Description
BIG_ENDIAN	Big endian integer data of the appropriate size (one, two, four, or eight bytes) and big endian IEEE floating-point formats for REAL and COMPLEX single- and double- and extended-precision numbers. INTEGER(KIND=1) data is the same for little endian and big endian.
CRAY	Big endian integer data of the appropriate size (one, two, four, or eight bytes) and big endian CRAY proprietary floating-point format for REAL and COMPLEX single- and double-precision numbers.
FDX	Little endian integer data of the appropriate size (one, two, four, or eight bytes) and the following little endian proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
FGX	Little endian integer data of the appropriate size (one, two, four, or eight bytes) and and the following little endian proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16)
IBM	Big endian integer data of the appropriate INTEGER size (one, two, or four bytes) and big endian IBM proprietary (System\370 and similar) floating-point format for REAL and COMPLEX single- and double-precision numbers.
LITTLE_ENDIAN	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following native little endian IEEE floating-point formats: • S_float for REAL (KIND=4) and COMPLEX (KIND=4) • T_float for REAL (KIND=8) and COMPLEX (KIND=8)

Keyword	Description
	• IEEE style X_float for REAL (KIND=16) and COMPLEX (KIND=16) For additional information on supported ranges for these data types, see Native IEEE Floating-Point Representations.
NATIVE	No conversion occurs between memory and disk. This is the default for unformatted files.
VAXD	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX D_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)
VAXG	Native little endian integers of the appropriate INTEGER size (one, two, four, or eight bytes) and the following little endian VAX proprietary floating-point formats: • VAX F_float for REAL (KIND=4) and COMPLEX (KIND=4) • VAX G_float for REAL (KIND=8) and COMPLEX (KIND=8) • VAX H_float for REAL (KIND=16) and COMPLEX (KIND=16)

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message is displayed.

See also:

Environment Variable F_UFMTENDIAN Method

Porting Nonnative Data

Keep this information in mind when porting nonnative data:

- When porting source code along with the unformatted data, vendors might use different units for specifying the record length (RECL specifier) of unformatted files. While formatted files are specified in units of characters (bytes), unformatted files are specified in longword units for Intel Fortran (default) and some other vendors.

To allow you to specify the RECL units (bytes or longwords) for unformatted files without source file modification, use the `-assume byterecl` (Linux OS and Mac OS X) or `/assume:byterecl` (Windows) compiler option. Alternatively, for Windows OS, you can specify Use Bytes as RECL=Unit for Unformatted Files in the Data Options property page.

The Fortran 95 standard (American National Standard Fortran 95, ANSI X3J3/96-007, and International Standards Organization standard ISO/IEC 1539-1:1997) states: "If the file is being connected for unformatted input/output, the length is measured in processor-dependent units."

- Certain vendors apply different `OPEN` statement defaults to determine the record type. The default record type (RECORDTYPE) with Intel Fortran depends on the values for the ACCESS and FORM specifiers for the OPEN statement.
- Certain vendors use a different identifier for the logical data types, such as hex FF instead of 01 to denote "true."
- Source code being ported may be coded specifically for big endian use.

Specifying the Data Format

Methods of Specifying the Data Format

There are a number of methods for specifying a nonnative numeric format for unformatted data:

- Setting an environment variable for a specific unit number before the file is opened. The environment variable is named FORT_CONVERT*n*, where *n* is the unit number. See Environment Variable FORT_CONVERTn Method.
- Setting an environment variable for a specific file name extension before the file is opened. The environment variable is named FORT_CONVERT.*ext* or FORT_CONVERT_*ext*, where *ext* is the file name extension (suffix). See Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method.
- Setting an environment variable for a set of units before the application is executed. The environment variable is named F_UFMTENDIAN. See Environment Variable F_UFMTENDIAN Method.
- Specifying the CONVERT keyword in the OPEN statement for a specific unit number. See OPEN Statement CONVERT Method.
- Compiling the program with an OPTIONS statement that specifies the CONVERT=*keyword* qualifier. This method affects all unit numbers using unformatted data specified by the program. See OPTIONS Statement Method.
- Compiling the program with the appropriate compiler option, which affects all unit numbers that use unformatted data specified by the program. Use the `-convert` *keyword* option for Linux* OS and Mac OS* X or, for Windows* OS, the command-line `/convert:`*keyword* option or the IDE equivalent.

If none of these methods are specified, the native LITTLE_ENDIAN format is assumed (no conversion occurs between disk and memory).

Any keyword listed in Supported Native and Nonnative Numeric Formats can be used with any of these methods, except for the Environment Variable F_UFMTENDIAN Method, which supports only LITTLE_ENDIAN and BIG_ENDIAN.

If you specify more than one method, the order of precedence when you open a file with unformatted data is to:

1. Check for an environment variable (FORT_CONVERT*n*) for the specified unit number (applies to any file opened on a particular unit).
2. Check for an environment variable (FORT_CONVERT.*ext* is checked before FORT_CONVERT_*ext*) for the specified file name extension (applies to all files opened with the specified file name extension).
3. Check for an environment variable (F_UFMTENDIAN) for the specified unit number (or for all units).

NOTE. This environment variable is checked only when the application starts executing.

4. Check the OPEN statement CONVERT specifier.
5. Check whether an OPTIONS statement with a CONVERT=*keyword* qualifier was present when the program was compiled.
6. Check whether the compiler option `-convert` *keyword* (Linux OS and Mac OS X) or `/convert:`*keyword* (Windows OS) was present when the program was compiled.

Environment Variable FORT_CONVERT.ext or FORT_CONVERT_ext Method

You can use this method to specify a non-native numeric format for each specified file name extension (suffix). You specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit `OPEN` to one or more unformatted files. You can use the format FORT_CONVERT.*ext* or FORT_CONVERT_*ext* (where *ext* is the file extension or suffix). The FORT_CONVERT.*ext* environment variable is checked before FORT_CONVERT_*ext* environment variable (if *ext* is the same).

For example, assume you have a previously compiled program that reads numeric data from one file and writes to another file using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from a file with a .dat file extension and write that data in native little endian format to a file with a extension of .data. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format

(S_float and T_float) when read from file.dat, and then written without conversion in native little endian IEEE format to the file with a suffix of .data, assuming that environment variables FORT_CONVERT.DATA and FORT_CONVERT*n* (for that unit number) are not defined.

Without requiring source code modification or recompilation of a program, the following command sets the appropriate environment variables before running the program:

Linux:

```
setenv FORT_CONVERT.DAT BIG_ENDIAN
```

Windows:

```
set FORT_CONVERT.DAT=BIG_ENDIAN
```

The FORT_CONVERT*n* method takes precedence over this method. When the appropriate environment variable is set when you open the file, the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable is used if a FORT_CONVERT*n* environment variable is not set for the unit number.

The FORT_CONVERT*n* and the FORT_CONVERT.*ext* or FORT_CONVERT_*ext* environment variable methods take precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

You can set the appropriate environment variable using the format FORT_CONVERT.*ext* or FORT_CONVERT_*ext*. If you also use Intel Fortran on Linux* systems, consider using the FORT_CONVERT_*ext* form, because a dot (.) cannot be used for environment variable names on certain Linux command shells. If you do define both FORT_CONVERT.*ext* and FORT_CONVERT_*ext* for the same extension (*ext*), the file defined by FORT_CONVERT.*ext* is used.

On Windows systems, the file name extension (suffix) is not case-sensitive. The extension must be part of the file name (not the directory path).

Environment Variable FORT_CONVERT*n* Method

You can use this method to specify a non-native numeric format for each specified unit number. You specify the numeric format at run time by setting the appropriate environment variable before an implicit or explicit `OPEN` to that unit number.

When the appropriate environment variable is set when you open the file, the environment variable is always used because this method takes precedence over the other methods. For instance, you might use this method to specify that a unit number will use a particular format instead of the format specified in the program (perhaps for a one-time file conversion).

For example, assume you have a previously compiled program that reads numeric data from unit 28 and writes it to unit 29 using unformatted I/O statements. You want the program to read nonnative big endian (IEEE floating-point) format from unit 28 and write that data in

native little endian format to unit 29. In this case, the data is converted from big endian IEEE format to native little endian IEEE memory format when read from unit 28, and then written without conversion in native little endian IEEE format to unit 29.

Without requiring source code modification or recompilation of this program, the following command sequence sets the appropriate environment variables before running a program.

Linux:

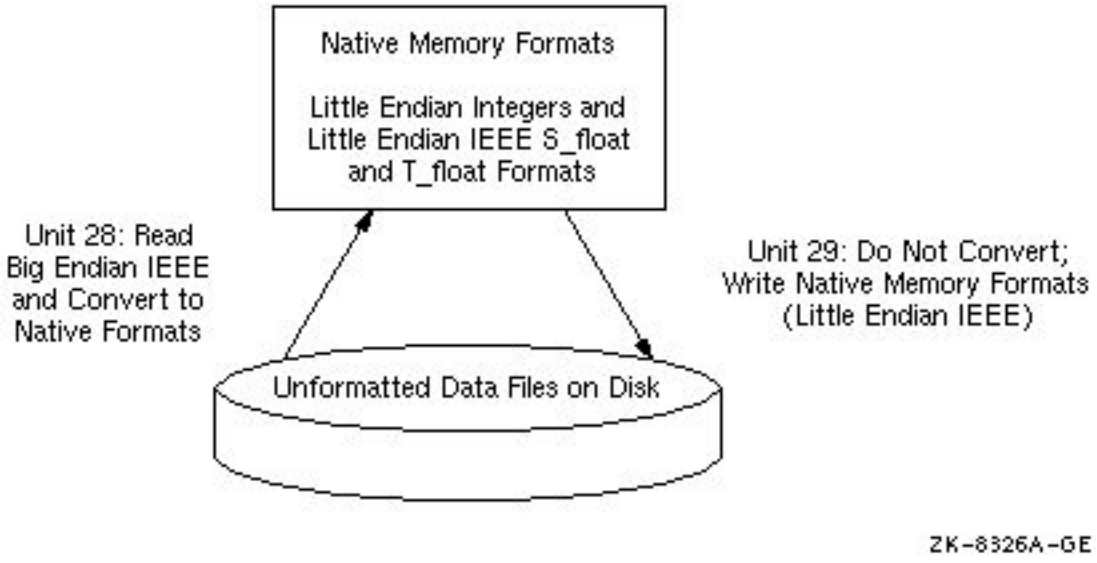
```
setenv FORT_CONVERT28 BIG_ENDIAN
setenv FORT_CONVERT29 NATIVE
```

Windows:

```
set FORT_CONVERT28=BIG_ENDIAN
set FORT_CONVERT29=NATIVE
```

The following figure shows the data formats used on disk and in memory when the program is run after the environment variables are set.

Figure 11: Sample Unformatted File Conversion



This method takes precedence over other methods.

Environment Variable F_UFMTENDIAN Method

This little-endian-big-endian conversion feature is intended for Fortran unformatted input/output operations. It enables the development and processing of files with little-endian and big-endian data organization.

The F_UFMTENDIAN environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units or for all units continues for the rest of the execution.

Specify the numbers of the units to be used for conversion purposes by setting `F_UFMTENDIAN`. Then, the `READ`/`WRITE` statements that use these unit numbers will perform relevant conversions. Other `READ`/`WRITE` statements will work in the usual way.

General Syntax for F_UFMTENDIAN

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

where:

```
MODE = big | little
EXCEPTION = big:ULIST | little:ULIST | ULIST
ULIST = U | ULIST,U
U = decimal | decimal -decimal
```

- `MODE` defines current format of data, represented in the files; it can be omitted.

  The keyword `little` means that the data has little endian format and will not be converted.This is the default.

  The keyword `big` means that the data has big endian format and will be converted.

- `EXCEPTION` is intended to define the list of exclusions for `MODE`.`EXCEPTION` keyword (`little` or `big`) defines data format in the files that are connected to the units from the `EXCEPTION` list. This value overrides `MODE` value for the units listed.

  The `EXCEPTION` keyword and the colon can be omitted. The default when the keyword is omitted is `big`.

- Each list member `U` is a simple unit number or a number of units. The number of list members is limited to 64.
- `decimal` is a non-negative decimal number less than 232.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Error messages may be issued during the little-endian-to-big-endian conversion. They are all fatal.

On Linux* systems, the command line for the variable setting in the shell is:

```
Sh: export F_UFMTENDIAN=MODE;EXCEPTION
```

NOTE. The environment variable value should be enclosed in quotes if the semicolon is present.

The environment variable can also have the following syntax:

```
F_UFMTENDIAN=u[,u] . . .
```

Examples

1. `F_UFMTENDIAN=big`

   All input/output operations perform conversion from `big-endian` to `little-endian` on `READ` and from little-endian to `big-endian` on `WRITE`.

2. `F_UFMTENDIAN="little;big:10,20"`

   or `F_UFMTENDIAN=big:10,20`

   or `F_UFMTENDIAN=10,20`

   The input/output operations perform big-endian to little endian conversion only on unit numbers 10 and 20.

3. `F_UFMTENDIAN="big;little:8"`

   No conversion operation occurs on unit number 8. On all other units, the input/output operations perform big-endian to little-endian conversion.

4. `F_UFMTENDIAN=10-20`

   The input/output operations perform big-endian to little-endian conversion on units 10, 11, 12 , ... 19, 20.

5. Assume you set `F_UFMTENDIAN=10,100` and run the following program.

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789
```

```
C prepare a little endian representation of data

open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)

C prepare a big endian representation of data

open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)

C read big endian data and operate with them on
C  little endian machine.

open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4

C Any operation with data, which have been read

C . . .
close(100)
stop
end
```

Now compare `lit.tmp` and `big.tmp` files with the help of the `od` utility:

```
> od -t x4 lit.tmp
0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034
> od -t x4 big.tmp
0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

You can see that the byte order is different in these files.

OPEN Statement CONVERT Method

You can use this method to specify a non-native numeric format for each specified unit number. This method requires an explicit file OPEN statement to specify the numeric format of the file for that unit number.

This method takes precedence over the OPTIONS statement and the compiler option `-convert keyword` (Linux OS and Mac OS X) or `/convert:keyword` (Windows OS) method, but has a lower precedence than the environment variable methods.

For example, the following source code shows how the OPEN statement would be coded to read unformatted VAXD numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20 (the absence of the CONVERT keyword or environment variables `FORT_CONVERT20`, `FORT_CONVERT.dat`, or `FORT_CONVERT_dat` indicates native little endian data for unit 20):

```
OPEN (CONVERT='VAXD', FILE='graph3.dat', FORM='UNFORMATTED', UNIT=15)
  .
  .
  .
OPEN (FILE='graph3_t.dat', FORM='UNFORMATTED', UNIT=20)
```

A hard-coded OPEN statement CONVERT keyword value cannot be changed after compile time. However, to allow selection of a particular format at run time, equate the CONVERT keyword to a variable and provide the user with a menu that allows selection of the appropriate format (menu choice sets the variable) before the `OPEN` occurs.

You can also select a particular format at run time for a unit number by using one of the environment variable methods (FORT_CONVERTn, FORT_CONVERT.ext or FORT_CONVERT_ext, or F_UFMTENDIAN), which take precedence over the `OPEN` statement CONVERT *`keyword`* method.

OPTIONS Statement Method

You can only specify one numeric file format for all unformatted file unit numbers using this method unless you also use one of the environment variable methods or `OPEN` statement CONVERT keyword method.

You specify the numeric format at compile time and must compile all routines under the same OPTIONS statement /CONVERT=*`keyword`* qualifier. You could use one source program and compile it using different ifort commands to create multiple executable programs that each read a certain format.

The environment variable methods and the `OPEN` statement CONVERT *`keyword`* method take precedence over this method. For instance, you might use the FORT_CONVERT*n* environment variable or `OPEN` CONVERT *`keyword`* method to specify each unit number that will use a format other than that specified using the `ifort` *`option`* method.

This method takes precedence over the `ifort -convert` keyword (Linux OS and Mac OS X) or `/convert:`*`keyword`* (Windows OS) compiler option method.

You can use OPTIONS statements to specify the appropriate floating-point formats (in memory and in unformatted files) instead of using the corresponding `ifort` command qualifiers. For example, to use VAX F_floating, G_floating, and X_floating as the unformatted file format, specify the following OPTIONS statement:

```
OPTIONS /CONVERT=VAXG
```

Because this method affects all unit numbers, you cannot read data in one format and write it in another format, unless you use it in combination with one of the environment variable methods or the OPEN statement CONVERT keyword method to specify a different format for a particular unit number.

For more information, see the OPTIONS statement.

Compiler Option -convert or /convert Method

You can only specify one numeric format for all unformatted file unit numbers using the compiler option `-convert` (Linux OS and Mac OS X) or `/convert` (Windows OS) method unless you also use one (or more) of the previous methods.

You specify the numeric format at compile time and must compile all routines under the same `-convert`*keyword* (Linux OS and Mac OS X) or `/convert:`*keyword* (Windows OS) compiler option. You can use the same source program and compile it using different ifort commands (or the equivalent in the IDE) to create multiple executable programs that each read a certain format.

If you specify other methods, they take precedence over this method. For instance, you might use the environment variable or OPEN statement CONVERT keyword method to specify each unit number that will use a format different than that specified using the `-convert` *keyword* (Linux OS and Mac OS X) or `/convert:`*keyword* (Windows OS) compiler option method for all other unit numbers.

For example, the following command compiles program `file.for` to use VAX D_floating (and F_floating) floating-point data for all unit numbers (unless superseded by one of the other methods). Data is converted between the file format and the little endian memory format (little endian integers, S_floating, T_floating, and X_floating little endian IEEE* floating-point format). The created file, `vconvert.exe`, can then be run:

Linux OS and Mac OS X:

```
ifort file.for -o vconvert.exe -convert vaxd
```

Windows OS:

```
ifort file.for /convert:vaxd /link /out:vconvert.exe
```

Because this method affects all unformatted file unit numbers, you cannot read data in one format and write it in another file format using the `-convert keyword` (Linux OS and Mac OS X) or `/convert:`*keyword* (Windows OS) compiler option method alone. You can if you use it in combination with the environment variable methods or the OPEN statement CONVERT keyword method to specify a different format for a particular unit number.

For more information, see the following topic:

- convert compiler option

Fortran I/O

Devices and Files Overview

In Fortran's I/O system, data is stored and transferred among files. All I/O data sources and destinations are considered files.

Devices such as the screen, keyboard and printer are external files, as are data files stored on a device such as a disk.

Variables in memory can also act as a file on a disk, and are typically used to convert ASCII representations of numbers to binary form. When variables are used in this way, they are called internal files.

Topics covered in this section include the following:

- Logical Devices
- Types and Forms of I/O Statements
- File Organization, File Access and File Structure
- Internal Files and Scratch Files
- File Records, including Record Types, Length, Access, and Transfer
- Using I/O Statements such as OPEN, INQUIRE, and CLOSE

For information on techniques you can use to improve I/O performance, see Improving I/O Performance.

Logical Devices

Every file, internal or external, is associated with a logical device. You identify the logical device associated with a file by a unit specifier (UNIT=). The unit specifier for an internal file is the name of the character variable associated with it. The unit specifier for an external file is either a number you assign with the `OPEN` statement, a number preconnected as a unit specifier to a device, or an asterisk (*).

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. External unit specifiers that are preconnected to certain devices do not have to be opened. External units that you connect are disconnected when program execution terminates or when the unit is closed by a `CLOSE` statement.

A unit must not be connected to more than one file at a time, and a file must not be connected to more than one unit at a time. You can OPEN an already opened file but only to change some of the I/O options for the connection, not to connect an already opened file or unit to a different unit or file.

You must use a unit specifier for all I/O statements, except in the following six cases:

- ACCEPT, which always reads from standard input, unless the FOR_ACCEPT environment variable is defined.
- INQUIRE by file, which specifies the filename, rather than the unit with which the file is associated.
- PRINT, which always writes to standard output, unless the FOR_PRINT environment variable is defined.
- READ statements that contain only an I/O list and format specifier, which read from standard input (UNIT=5), unless the FOR_READ environment variable is defined.
- WRITE statements that contain only an I/O list and format specifier, which write to standard output, unless the FOR_PRINT environment variable is defined.
- TYPE, which always writes to standard output, unless the FOR_TYPE environment variable is defined.

External Files

A unit specifier associated with an external file must be either an integer expression or an asterisk (*). The integer expression must be in the range 0 (zero) to a maximum value of 2,147,483,640. (The predefined parameters FOR_K_PRINT_UNITNO, FOR_K_TYPE_UNITNO, FOR_K_ACCEPT_UNITNO, and FOR_K_READ_UNITNO may not be in that range. For more information, see the Language Reference.)

The following example connects the external file UNDAMP.DAT to unit 10 and writes to it:

```
OPEN (UNIT = 10, FILE = 'UNDAMP.DAT')
WRITE (10, '(A18,\)') ' Undamped Motion:'
```

The asterisk (*) unit specifier specifies the keyboard when reading and the screen when writing. The following example uses the asterisk specifier to write to the screen:

```
WRITE (*, '(1X, A30,\)') ' Write this to the screen.'
```

Intel Fortran has four units preconnected to external files (devices), as shown in the following table.

External Unit Specifier	Environment Variable	Description

Asterisk (*)	None	Always represents the keyboard and screen (unless the appropriate environment variable is defined, such as FOR_READ).
0	FORT0	Initially represents the screen (unless FORT0 is explicitly defined)
5	FORT5	Initially represents the keyboard (unless FORT5 is explicitly defined)
6	FORT6	Initially represents the screen (unless FORT6 is explicitly defined)

The asterisk (*) specifier is the only unit specifier that cannot be reconnected to another file, and attempting to close this unit causes a compile-time error. Units 0, 5, and 6, however, can be connected to any file with the `OPEN` statement. If you close unit 0, 5, or 6, it is automatically reconnected to its preconnected device the next time an I/O statement attempts to use that unit.

Intel® Fortran does not support buffering to `stdout` or `stdin`. All I/O to units * and 6 use line buffering. Therefore, C and Fortran output to stdout should work well as long as the C code is not performing buffering. In the case of buffering, the C code will have to flush the buffers with each write. For more information on `stdout` and `stdin`, see Assigning Files to Logical Units.

You can change these preconnected files by doing one of the following:

- Using an OPEN statement to open unit 5, 6, or 0. When you explicitly OPEN a file for unit 5, 6, or 0, the OPEN statement keywords specify the file-related information to be used instead of the preconnected standard I/O file.
- Setting the appropriate environment variable (`FORT`*`n`*) to redirect I/O to an external file.

To redirect input or output from the standard preconnected files at run time, you can set the appropriate environment variable or use the appropriate shell redirection character in a pipe (such as > or <).

When you omit the file name in the OPEN statement or use an implicit OPEN, you can define the environment variable FORT*n* to specify the file name for a particular unit number *n*. An exception to this is when the compiler option `-fpscomp filesfromcmd` (Linux* OS and Mac OS* X) or `/fpscomp:filesfromcmd` (Windows) is specified.

For example, if you want unit 6 to write to a file instead of standard output, set the environment variable FORT6 to the path and filename to be used before you run the program. If the appropriate environment variable is not defined, a default filename is used, in the form fort.*n* where *n* is the logical unit number.

The following example writes to the preconnected unit 6 (the screen), then reconnects unit 6 to an external file and writes to it, and finally reconnects unit 6 to the screen and writes to it:

```
     REAL a, b
!  Write to the screen (preconnected unit 6).
    WRITE(6, '('' This is unit 6'')')
!  Use the OPEN statement to connect unit 6
!  to an external file named 'COSINES'.
    OPEN (UNIT = 6, FILE = 'COSINES', STATUS = 'NEW')
    DO a = 0.1, 6.3, 0.1
         b = COS (a)
!  Write to the file 'COSINES'.
          WRITE (6, 100) a, b
100        FORMAT (F3.1, F5.2)
    END DO
!  Close it.
    CLOSE (6)
!  Reconnect unit 6 to the screen, by writing to it.
    WRITE(6,' ('' Cosines completed'')')
    END
```

Internal Files

The unit specifier associated with an internal file is a character string or character array. There are two types of internal files:

- An internal file that is a character variable, character array element, or noncharacter array element has exactly one record, which is the same length as the variable, array element, or noncharacter array element.
- An internal file that is a character array, a character derived type, or a noncharacter array is a sequence of elements, each of which is a record. The order of records is the same as the order of array elements or type elements, and the record length is the length of one array element or the length of the derived-type element.

Follow these rules when using internal files:

- Use only formatted I/O, including I/O formatted with a format specification and list-directed I/O. (List-directed I/O is treated as sequential formatted I/O.) Namelist formatting is not allowed.

- If the character variable is an allocatable array or array part of an allocatable array, the array must be allocated before use as an internal file. If the character variable is a pointer, it must be associated with a target.
- Use only READ and WRITE statements. You cannot use file connection (`OPEN`, `CLOSE`), file positioning (`REWIND`, `BACKSPACE`), or file inquiry (`INQUIRE`) statements with internal files.

You can read and write internal files with `FORMAT` I/O statements or list-directed I/O statements exactly as you can external files. Before an I/O statement is executed, internal files are positioned at the beginning of the variable, before the first record.

With internal files, you can use the formatting capabilities of the I/O system to convert values between external character representations and Fortran internal memory representations. That is, reading from an internal file converts the ASCII representations into numeric, logical, or character representations, and writing to an internal file converts these representations into their ASCII representations.

This feature makes it possible to read a string of characters without knowing its exact format, examine the string, and interpret its contents. It also makes it possible, as in dialog boxes, for the user to enter a string and for your application to interpret it as a number.

If less than an entire record is written to an internal file, the rest of the record is filled with blanks.

In the following example, `str` and `fname` specify internal files:

```
    CHARACTER(10) str
   INTEGER n1, n2, n3
   CHARACTER(14) fname
   INTEGER   i
   str = " 1   2   3"
! List-directed READ sets n1 = 1, n2 = 2, n3 = 3.
   READ(str, *) n1, n2, n3
   i = 4
! Formatted WRITE sets fname = 'FM004.DAT'.
   WRITE (fname, 200) i
200 FORMAT ('FM', I3.3, '.DAT')
```

Types of I/O Statements

The table below lists the Intel Fortran I/O statements:

Category and statement name	Description
File connection	

Category and statement name	Description
OPEN	Connects a unit number with an external file and specifies file connection characteristics.
CLOSE	Disconnects a unit number from an external file.
File inquiry	
DEFINE FILE	Specifies file characteristics for a direct access relative file and connects the unit number to the file, similar to an OPEN statement. Provided for compatibility with compilers older than FORTRAN-77.
INQUIRE	Returns information about a named file, a connection to a unit, or the length of an output item list.
Record position	
BACKSPACE	Moves the record position to the beginning of the previous record (sequential access only).
DELETE	Marks the record at the current record position in a relative file as deleted (direct access only).
ENDFILE	Writes an end-of-file marker after the current record (sequential access only).
FIND	Changes the record position in a direct access file. Provided for compatibility with compilers older than FORTRAN-77.
REWIND	Sets the record position to the beginning of the file (sequential access only).
Record input	
READ	Transfers data from an external file record or an internal file to internal storage.
ACCEPT	Reads input from stdin. Unlike READ, ACCEPT only provides formatted sequential input and does not specify a unit number.
Record output	

Category and statement name	Description
WRITE	Transfers data from internal storage to an external file record or to an internal file.
REWRITE	Transfers data from internal storage to an external file record at the current record position (direct access relative files only).
TYPE	Writes record output to stdout.
PRINT	Transfers data from internal storage to stdout. Unlike WRITE, PRINT only provides formatted sequential output and does not specify a unit number.
FLUSH	Flushes the contents of an external unit's buffer to its associated file.

In addition to the READ, WRITE, REWRITE, TYPE, and PRINT statements, other I/O record-related statements are limited to a specific file organization. For instance:

- The DELETE statement only applies to relative files. (Detecting deleted records is only available if the `-vms` option was specified when the program was compiled.)
- The BACKSPACE statement only applies to sequential files open for sequential access.
- The REWIND statement only applies to sequential files open for sequential access and to direct access files.
- The ENDFILE statement only applies to certain types of sequential files open for sequential access and to direct access files.

The file-related statements (OPEN, INQUIRE, and CLOSE) apply to any relative or sequential file.

Forms of I/O Statements

Each type of record I/O statement can be coded in a variety of forms. The form you select depends on the nature of your data and how you want it treated. When opening a file, specify the form using the FORM specifier.

The following are the forms of I/O statements:

- *Formatted I/O statements* contain explicit format specifiers that are used to control the translation of data from internal (binary) form within a program to external (readable character) form in the records, or vice versa.

- *List-directed* and *namelist I/O statements* are similar to formatted statements in function. However, they use different mechanisms to control the translation of data: formatted I/O statements use explicit format specifiers, and list-directed and namelist I/O statements use data types.
- *Unformatted I/O statements* do not contain format specifiers and therefore do not translate the data being transferred (important when writing data that will be read later).

Formatted, list-directed, and namelist I/O forms require translation of data from internal (binary) form within a program to external (readable character) form in the records. Consider using unformatted I/O for the following reasons:

- Unformatted data avoids the translation process, so I/O tends to be faster.
- Unformatted data avoids the loss of precision in floating-point numbers when the output data will subsequently be used as input data.
- Unformatted data conserves file storage space (stored in binary form).

To write data to a file using formatted, list-directed, or namelist I/O statements, specify FORM= 'FORMATTED' when opening the file. To write data to a file using unformatted I/O statements, specify FORM= 'UNFORMATTED' when opening the file.

Data written using formatted, list-directed, or namelist I/O statements is referred to as formatted data; data written using unformatted I/O statements is referred to as unformatted data.

When reading data from a file, you should use the same I/O statement form that was used to write the data to the file. For instance, if data was written to a file with a formatted I/O statement, you should read data from that file with a formatted I/O statement.

Although I/O statement form is usually the same for reading and writing data in a file, a program can read a file containing unformatted data (using unformatted input) and write it to a separate file containing formatted data (using formatted output). Similarly, a program can read a file containing formatted data and write it to a different file containing unformatted data.

You can access records in any sequential or relative file using sequential access. For relative files and certain (fixed-length) sequential files, you can also access records using direct access.

The table below shows the main record I/O statements, by category, that can be used in Intel Fortran programs.

File Type, Access, and I/O Form	Available Statements
External file, sequential access	
Formatted	READ, WRITE, PRINT, ACCEPT, TYPE, REWRITE
List-directed	READ, WRITE, PRINT, ACCEPT, TYPE

File Type, Access, and I/O Form	Available Statements
Namelist	READ, WRITE, PRINT, ACCEPT, TYPE
Unformatted	READ, WRITE, REWRITE
External file, direct access	
Formatted	READ, WRITE, REWRITE
Unformatted	READ, WRITE, REWRITE
External file, stream access	
Formatted	READ, WRITE
List-directed	READ, WRITE
Namelist	READ, WRITE
Unformatted	READ, WRITE
Internal file	
Formatted	READ, WRITE
List-directed	READ, WRITE
Unformatted	None

NOTE. You can use the REWRITE statement only for relative files, using direct access.

Assigning Files to Logical Units

Most I/O operations involve a disk file, keyboard, or screen display. Other devices can also be used:

- Sockets can be read from or written to if a USEROPEN routine (usually written in C) is used to open the socket.
- Pipes opened for read and write access block (wait until data is available) if you issue a READ to an empty pipe.

- Pipes opened for read-only access return EOF if you issue a READ to an empty pipe.

You can access the terminal screen or keyboard by using preconnected files listed in Logical Devices.

You can choose to assign files to logical units by using one of the following methods:

- Using default values, such as a preconnected unit
- Supplying a file name (and possibly a directory) in an OPEN statement
- Using environment variables

Using Default Values

In the following example, the PRINT statement is associated with a preconnected unit (stdout) by default.

```
PRINT *,100
```

The READ statement associates the logical unit 7 with the file `fort.7` (because the FILE specifier was omitted) by default:

```
OPEN (UNIT=7,STATUS='NEW')
READ (7,100)
```

Supplying a File Name in an OPEN Statement

The FILE specifier in an OPEN statement typically specifies only a file name (such as `filnam`) or contains both a directory and file name (such as `/usr/proj/filnam`).

For example:

```
OPEN (UNIT=7, FILE='FILNAM.DAT', STATUS='OLD')
```

The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).

Implied OPEN

Performing an implied OPEN means that the FILE and DEFAULTFILE specifier values are not specified and an environment variable is used, if present. Thus, if you used an implied OPEN, or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Using Environment Variables

You can use shell commands to set the appropriate environment variable to a value that indicates a directory (if needed) and a file name to associate a unit with an external file.

Intel Fortran recognizes environment variables for each logical I/O unit number in the form of FORT`n`, where `n` is the logical I/O unit number. If a file name is not specified in the OPEN statement and the corresponding FORT`n` environment variable is not set for that unit number, Intel Fortran generates a file name in the form fort.`n`, where `n` is the logical unit number.

Implied Intel Fortran Logical Unit Numbers

The ACCEPT, PRINT, and TYPE statements, and the use of an asterisk (*) in place of a unit number in READ and WRITE statements, do not include an explicit logical unit number.

Each of these Fortran statements uses an implicit internal logical unit number and environment variable. Each environment variable is in turn associated by default with one of the Fortran file names that are associated with standard I/O files. The table below shows these relationships:

Intel Fortran statement	Environment variable	Standard I/O file name
READ (*,f) iolist	FOR_READ	stdin
READ f,iolist	FOR_READ	stdin
ACCEPT f,iolist	FOR_ACCEPT	stdin
WRITE (*,f) iolist	FOR_PRINT	stdout
PRINT f,iolist	FOR_PRINT	stdout
TYPE f,iolist	FOR_TYPE	stdout
WRITE(0,f) iolist	FORT0	stderr
READ(5,f) iolist	FORT5	stdin
WRITE(6,f) iolist	FORT6	stdout

You can change the file associated with these Intel Fortran environment variables, as you would any other environment variable, by means of the environment variable assignment command. For example:

```
setenv FOR_READ /usr/users/smith/test.dat
```

After executing the preceding command, the environment variable for the READ statement using an asterisk refers to file `test.dat` in the specified directory.

NOTE. The association between the logical unit number and the physical file can occur at run-time. Instead of changing the logical unit numbers specified in the source program, you can change this association at run time to match the needs of the program and the available resources. For example, before running the program, a script file can set the appropriate environment variable or allow the terminal user to type a directory path, file name, or both.

File Organization

File organization refers to the way records are physically arranged on a storage device. This topic describes the two main types of file organization.

Related topics describe the following:

- *Record type* refers to whether records in a file are all the same length, are of varying length, or use other conventions to define where one record ends and another begins. For more information on record types, see Record Types.
- *Record access* refers to the method used to read records from or write records to a file, regardless of its organization. The way a file is organized does not necessarily imply the way in which the records within that file will be accessed. For more information on record access, see File Access and File Structure and Record Access.

Types of File Organization

Fortran supports two types of file organizations:

- Sequential
- Relative

The organization of a file is specified by means of the ORGANIZATION keyword in the `OPEN` statement.

The default file organization is always ORGANIZATION= 'SEQUENTIAL' for an OPEN statement.

You can store sequential files on magnetic tape or disk devices, and can use other peripheral devices, such as terminals, pipes, and line printers as sequential files.

You must store relative files on a disk device.

Sequential File Organization

A sequentially organized file consists of records arranged in the sequence in which they are written to the file (the first record written is the first record in the file, the second record written is the second record in the file, and so on). As a result, records can be added only at the end of the file. Attempting to add records at some place other than the end of the file will result in the file begin truncated at the end of the record just written.

Sequential files are usually read sequentially, starting with the first record in the file. Sequential files with a fixed-length record type that are stored on disk can also be accessed by relative record number (direct access).

Relative File Organization

Within a relative file are numbered positions, called *cells*. These cells are of fixed equal length and are consecutively numbered from 1 to *n*, where 1 is the first cell, and *n* is the last available cell in the file. Each cell either contains a single record or is empty. Records in a relative file are accessed according to cell number. A cell number is a record's relative record number; its location relative to the beginning of the file. By specifying relative record numbers, you can directly retrieve, add, or delete records regardless of their locations. (Detecting deleted records is only available if you specified the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) option when the program was compiled.)

When creating a relative file, use the RECL value to determine the size of the fixed-length cells. Within the cells, you can store records of varying length, as long as their size does not exceed the cell size.

Internal Files and Scratch Files

Intel Fortran supports two other types of files:

- Internal files
- Scratch files

Internal Files

When you use sequential access, you can use an internal file to reference character data in a buffer. The transfer occurs between internal storage and internal storage (unlike external files), such as between user variables and a character array.

An internal file consists of any of the following:

- Character variable
- Character-array element
- Character array
- Character substring
- Character array section without a vector subscript

Instead of specifying a unit number for the READ or WRITE statement, use an internal file specifier in the form of a character scalar memory reference or a character-array name reference.

An internal file is a designated internal storage space (variable buffer) of characters that is treated as a sequential file of fixed-length records. To perform internal I/O, use formatted and list-directed sequential READ and WRITE statements. You cannot use file-related statements such as OPEN and INQUIRE on an internal file (no unit number is used).

If an internal file is made up of a single character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the character variable, array element, or substring it contains. If an internal file is made up of a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined (a value has been assigned to the record).

Prior to each READ and WRITE statement, an internal file is always positioned at the beginning of the first record.

Scratch Files

Scratch files are created by specifying STATUS= 'SCRATCH' in an OPEN statement. By default, these temporary files are created in (and later deleted from) the directory specified in the OPEN statement DEFAULTFILE (if specified).

File Access and File Structure

Fortran supports three methods of file access:

- Sequential
- Direct
- Stream

Fortran supports three kinds of file structure:

- Formatted
- Unformatted
- Binary

Sequential-access and direct-access files can have any of the three file structures. Stream-access files can have a file structure of formatted or unformatted.

Choosing a File Access and File Structure

Each kind of file has advantages and the best choice depends on the application you are developing:

- Formatted Files

  You create a formatted file by opening it with the FORM='FORMATTED' option, or by omitting the FORM parameter when creating a sequential file. The records of a formatted file are stored as ASCII characters; numbers that would otherwise be stored in binary form are converted to ASCII format. Each record ends with the ASCII carriage return (CR) and/or line feed (LF) characters.

  If you need to view a data file's contents, use a formatted file. You can load a formatted file into a text editor and read its contents directly, that is, the numbers would look like numbers and the strings like character strings, whereas an unformatted or binary file looks like a set of hexadecimal characters.

- Unformatted Files

  You create an unformatted file by opening it with the FORM='UNFORMATTED' option, or by omitting the FORM parameter when creating a direct-access file. An unformatted file is a series of records composed of physical blocks. Each record contains a sequence of values stored in a representation that is close to that used in program memory. Little conversion is required during input/output.

  The lack of formatting makes these files quicker to access and more compact than files that store the same information in a formatted form. However, if the files contain numbers, you will not be able to read them with a text editor.

- Binary Files

  You create a binary file by specifying FORM='BINARY'. Binary files are similar to unformatted files, except binary files have no internal record format associated with them.

- Sequential-Access Files

Data in sequential files must be accessed in order, one record after the other (unless you change your position in the file with the REWIND or BACKSPACE statements). Some methods of I/O are possible only with sequential files, including nonadvancing I/O, list-directed I/O, and namelist I/O. Internal files also must be sequential files. You must use sequential access for files associated with sequential devices.

A sequential device is a physical storage device that does not allow explicit motion (other than reading or writing). The keyboard, screen, and printer are all sequential devices.

- Direct-Access Files

  Data in direct-access files can be read or written to in any order. Records are numbered sequentially, starting with record number 1. All records have the length specified by the RECL option in the OPEN statement. Data in direct files is accessed by specifying the record you want within the file. If you need random access I/O, use direct-access files. A common example of a random-access application is a database.

- Stream-Access Files

  Stream access I/O is a method of accessing a file without reference to a record structure. With stream access, a file is seen as a continuous sequence of bytes and is addressed by a positive integer starting from 1.

  To enable stream access, specify ACCESS='STREAM' in the OPEN statement for the file. You can use the STREAM= specifier in the INQUIRE statement to determine if STREAM is listed in the set of possible access methods for the file. A value of YES, NO, or UNKNOWN is returned.

  A file enabled for stream access is positioned by file storage units (normally bytes) starting at position 1. To determine the current position, use the POS= specifier in an INQUIRE statement for the unit. You can indicate a required position in a read or write statement with a POS= specifier.

  Stream access can be either formatted or unformatted.

  When connected for formatted stream access, an external file has the following characteristics:

  - The first file storage unit in the file is at position 1. The relationship between positions of successive file storage units is processor dependent; not all positive integers need to correspond to valid positions.
  - Some file storage units may contain record markers; this imposes a record structure on the file in addition to its stream structure. If there is no record marker at the end of the file, the final record is incomplete. Writing an empty record with no record marker has no effect.

When connected for unformatted stream access, an external file has the following characteristics:

- The first file storage unit in the file is at position 1. The position of each subsequent file storage unit is one greater than that of the preceding file storage unit.
- If it is possible to position the file, the file storage units do not need to be read or written in order of their position. For example, you may be able to write the file storage unit at position 2, even though the file storage unit at position 1 has not been written.
- Any file storage unit may be read from the file while it is connected to a unit, provided that the file storage unit has been written since the file was created, and a READ statement for this connection is allowed.
- You cannot use BACKSPACE in an unformatted stream.

File Records

Files may be composed of records. Each record is one entry in the file. It can be a line from a terminal or a logical record on a magnetic tape or disk file. All records within one file are of the same type.

In Fortran, the number of bytes transferred to a record must be less than or equal to the record length. One record is transferred for each unformatted `READWRITE` or `READ` statement. A formatted `WRITE` or statement can transfer more than one record using the slash (`/`) edit descriptor.

For binary files, a single `READ` or `WRITE` statement reads or writes as many records as needed to accommodate the number of bytes being transferred. On output, incomplete formatted records are padded with spaces. Incomplete unformatted and binary records are padded with undefined bytes (zeros).

Record Types

An I/O record is a collection of data items, called fields, that are logically related and are processed as a unit. The record type refers to the convention for storing fields in records.

The record type of the data within a file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

A number of record types are available, as shown in the following table. The table also lists the record overhead. This refers to bytes associated with each record that are used internally by the file system and are not available when a record is read or written. Knowing the record overhead helps when estimating the storage requirements for an application. Although the overhead bytes exist on the storage media, do not include them when specifying the record length with the RECL specifier in an OPEN statement.

Record Type	Available File Organizations and Portability Considerations	Record Overhead
Fixed-length	Relative or sequential file organizations.	None for sequential or for relative if the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) option was omitted. One byte for relative if the `vms` option was specified.
Variable-length	Sequential file organization only. The variable-length record type is generally the most portable record type across multi-vendor platforms.	Eight bytes per record.
Segmented	Sequential file organization only and only for unformatted sequential access. The segmented record type is unique to Intel Fortran and should not be used for portability with programs written in languages other than Fortran or for places where Intel Fortran is not used. However, because the segmented record type is unique to Intel Fortran products, formatted data in segmented files can be ported across Intel Fortran platforms.	Four bytes per record. One additional padding byte (space) is added if the specified record size is an odd number.
Stream (uses no record terminator)	Sequential file organization only.	None required.
Stream_CR (uses CR as record terminator)	Sequential file organization only.	One byte per record.
Stream_LF (uses LF as record terminator)	Sequential file organization only.	One byte per record.

Record Type	Available File Organizations and Portability Considerations	Record Overhead
Stream_CRLF (uses CR and LF as record terminator)	Sequential file organization only.	Two bytes per record.

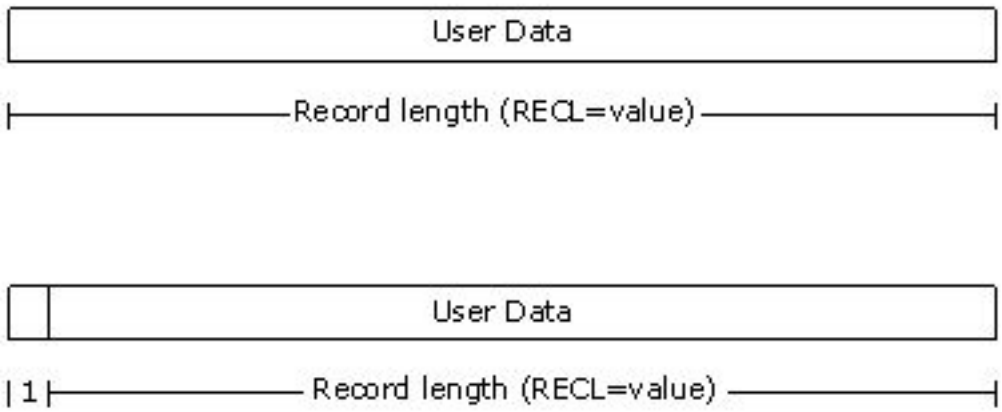
Fixed-Length Records

When you specify fixed-length records, you are specifying that all records in the file contain the same number of bytes. When you open a file that is to contain fixed-length records, you must specify the record size using the RECL keyword. A sequentially organized opened file for direct access must contain fixed-length records, to allow the record position in the file to be computed correctly.

For relative files, the layout and overhead of fixed-length records depends upon whether the program accessing the file was compiled using the `-vms` (Linux OS and Mac OS X) or `/vms` (Windows OS) option:

- For relative files where the vms option was omitted (the default), each record has no control information.
- For relative files where the vms option was specified, each record has one byte of control information at the beginning of the record.

The following figures show the record layout of fixed-length records. The first is for all sequential and relative files where the `vms` option was omitted. The second is for relative files where the `vms` option was specified.



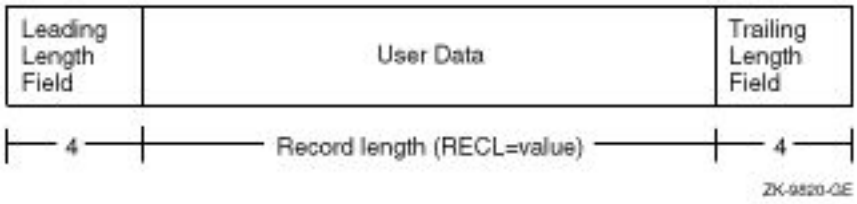
Variable-Length Records

Variable-length records can contain any number of bytes up to a specified maximum record length, and apply only to sequential files.

Variable-length records are prefixed and suffixed by 4 bytes of control information containing length fields. The trailing length field allows a BACKSPACE request to skip back over records efficiently. The 4-byte integer value stored in each length field indicates the number of data bytes (excluding overhead bytes) in that particular variable-length record.

The character count field of a variable-length record is available when you read the record by issuing a READ statement with a Q format descriptor. You can then use the count field information to determine how many bytes should be in an I/O list.

The record layout of variable-length records that are less than 2 gigabytes is shown below:

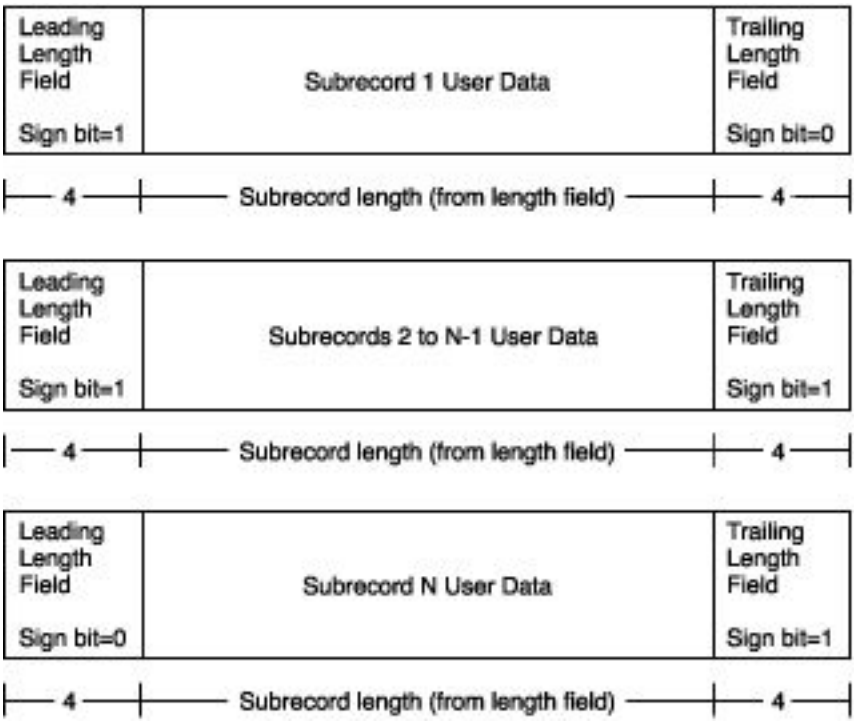


For a record length greater than 2,147,483,639 bytes, the record is divided into *subrecords*. The subrecord can be of any length from 1 to 2,147,483,639, inclusive.

The sign bit of the leading length field indicates whether the record is continued or not. The sign bit of the trailing length field indicates the presence of a preceding subrecord. The position of the sign bit is determined by the endian format of the file.

A subrecord that is continued has a leading length field with a sign bit value of 1. The last subrecord that makes up a record has a leading length field with a sign bit value of 0. A subrecord that has a preceding subrecord has a trailing length field with a sign bit value of 1. The first subrecord that makes up a record has a trailing length field with a sign bit value of 0. If the value of the sign bit is 1, the length of the record is stored in twos-complement notation.

The record layout of variable-length records that are greater than 2 gigabytes is shown below:



Files written with variable-length records by Intel Fortran programs usually cannot be accessed as text files. Instead, use the Stream_LF record format for text files with records of varying length.

Segmented Records

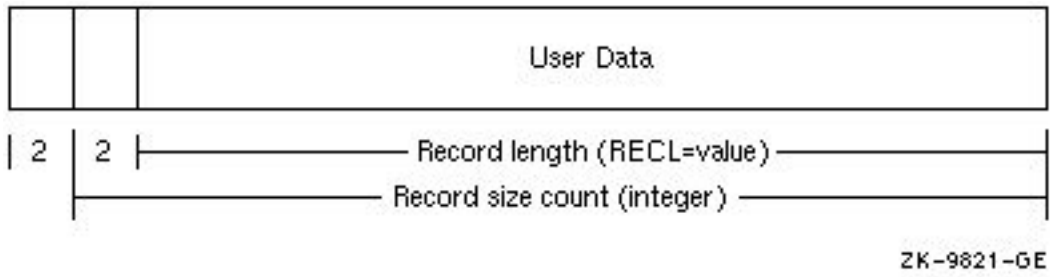
A segmented record is a single logical record consisting of one or more variable-length, unformatted records in a sequentially organized disk file. Unformatted data written to sequentially organized files using sequential access is stored as segmented records by default.

Segmented records are useful when you want to write exceptionally long records but cannot or do not wish to define one long variable-length record, perhaps because virtual memory limitations can prevent program execution. By using smaller, segmented records, you reduce the chance of problems caused by virtual memory limitations on systems on which the program may execute.

For disk files, the segmented record is a single logical record that consists of one or more segments. Each segment is a physical record. A segmented (logical) record can exceed the absolute maximum record length (2.14 billion bytes), but each segment (physical record) individually cannot exceed the maximum record length.

To access an unformatted sequential file that contains segmented records, specify FORM='UNFORMATTED' and RECORDTYPE='SEGMENTED' when you open the file.

As shown below, the layout of segmented records consists of 4 bytes of control information followed by the user data.



The control information consists of a 2-byte integer record length count (includes the 2 bytes used by the segment identifier), followed by a 2-byte integer segment identifier that identifies this segment as one of the following:

Identifier Value	Segment Identified
0	One of the segments between the first and last segments
1	First segment
2	Last segment
3	Only segment

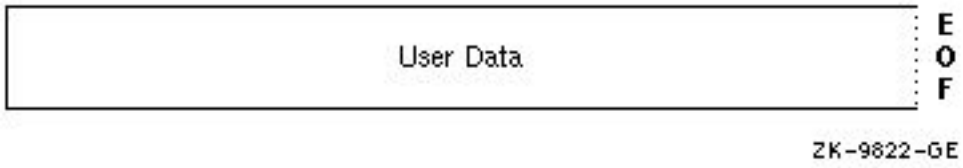
If the specified record length is an odd number, the user data will be padded with a single blank (one byte), but this extra byte is not added to the 2-byte integer record size count.

Avoid the segmented record type when the application requires that the same file be used for programs written in languages other than Intel Fortran and for non-Intel platforms.

Stream File Data

A stream file is not grouped into records and contains no control information. Stream files are used with CARRIAGECONTROL='NONE' and contain character or binary data that is read or written only to the extent of the variables specified on the input or output statement.

The layout of a stream file appears below.



Stream_CR, Stream_LF and Stream_CRLF Records

Stream_CR, Stream_LF, and Stream_CRLF records are variable-length records whose length is indicated by explicit record terminators embedded in the data, not by a count. These terminators are automatically added when you write records to a stream-type file and are removed when you read records.

Each variety uses either a different 1-byte or 2-byte record terminator:

- Stream_CR files use only a carriage-return as the terminator, so Stream_CR files must not contain embedded carriage-return characters.
- Stream_LF files use only a line-feed (new line) as the terminator, so Stream_LF files must not contain embedded line-feed (new line) characters. This is the usual operating system text file record type on Linux* OS and Mac OS* X systems.
- Stream_CRLF files use a carriage return/line-feed (new line) pair as the terminator, so Stream_CRLF files must not contain embedded carriage returns or line-feed (new line) characters. This is the usual operating system text file record type on Windows* systems.

Guidelines for Choosing a Record Type

Before you choose a record type, consider whether your application will use formatted or unformatted data. If you are using formatted data, you can choose any record type except segmented. If you are using unformatted data, avoid the Stream, Stream_CR, Stream_LF and Stream_CRLF record types.

The segmented record type can only be used for unformatted sequential access with sequential files. You should not use segmented records for files that are read by programs written in languages other than Intel Fortran.

The Stream, Stream_CR, Stream_LF, Stream_CRLF and segmented record types can be used only with sequential files.

The default record type (RECORDTYPE) depends on the values for the ACCESS and FORM specifiers for the OPEN statement. (The RECORDTYPE= specifier is ignored for stream access.)

The record type of the file is not maintained as an attribute of the file. The results of using a record type other than the one used to create the file are indeterminate.

An I/O record is a collection of fields (data items) that are logically related and are usually processed as a unit.

Unless you specify nonadvancing I/O (ADVANCE specifier), each Intel Fortran I/O statement transfers at least one record.

Specifying the Line Terminator for Formatted Files

Use the FOR_FMT_TERMINATOR environment variable to specify the line terminator value used for Fortran formatted files with no explicit RECORDTYPE= specifier.

The FOR_FMT_TERMINATOR environment variable is processed once at the beginning of program execution. Whatever it specifies for specific units continues for the rest of the execution.

You can specify the numbers of the units to have a specific record terminator. The READ/WRITE statements that use these unit numbers will now use the specified record terminators. Other READ/WRITE statements will work in the usual way.

A RECORDTYPE=specifier on an OPEN statement overrides the value set by FOR_FMT_TERMINATOR. The FOR_FMT_TERMINATOR value is ignored for ACCESS='STREAM' files.

General Syntax for FOR_FMT_TERMINATOR

The syntax for this environment variable is as follows:

```
FOR_FMT_TERMINATOR=MODE[:ULIST][;MODE[:ULIST]]
```

where:

```
MODE=CR | LF | CRLF
ULIST = U | ULIST,U
U = decimal | decimal - decimal
```

- `MODE` specifies the record terminator to be used. The keyword `CR` means to terminate records with a carriage return. The keyword `LF` means to terminate records with a line feed; this is the default on Linux OS and Mac OS X systems. The keyword `CRLF` means to terminate records with a carriage return/line feed pair; this is the default on Windows systems.
- Each list member `U` is a simple unit number or a number of units as a range. The number of list members is limited to 64.
- `decimal` is a non-negative decimal number less than 232.

No spaces are allowed inside the FOR_FMT_TERMINATOR value.

On Linux* systems, the command line for the variable setting in the shell is:

```
Sh: export FOR_FMT_TERMINATOR=MODE:ULIST
```

NOTE. The environment variable value should be enclosed in quotes if the semicolon is present.

Example:

The following specifies that all input/output operations on unit numbers 10, 11, and 12 have records terminated with a carriage return/line feed pair:

```
FOR_FMT_TERMINATOR=CRLF:10-12
```

Record Length

Use the RECL specifier to specify the record length.

The units used for specifying record length depend on the form of the data:

- Formatted files (FORM= 'FORMATTED'): Specify the record length in bytes.
- Unformatted files (FORM= 'UNFORMATTED'): Specify the record length in 4-byte units, unless you specify the `-assume byterecl` (Linux OS and Mac OS X) or `/assume:byterecl` (Windows OS) option to request 1-byte units.

For all but variable-length sequential records on 64-bit addressable systems, the maximum record length is 2.147 billion bytes (2,147,483,647 minus the bytes for record overhead). For variable-length sequential records on 64-bit addressable systems, the theoretical maximum record length is about 17,000 gigabytes. When considering very large record sizes, also consider limiting factors such as system virtual memory.

NOTE. The RECL specifier is ignored for stream access.

Record Access

Record access refers to how records will be read from or written to a file, regardless of the file's organization. Record access is specified each time you open a file; it can be different each time. The type of record access permitted is determined by the combination of file organization and record type.

For instance, you can:

- Add records to a sequential file with ORGANIZATION= 'SEQUENTIAL' and POSITION= 'APPEND' (or use ACCESS= 'APPEND').
- Add records sequentially by using multiple WRITE statements, close the file, and then open it again with ORGANIZATION= 'SEQUENTIAL' and ACCESS= 'SEQUENTIAL' (or ACCESS= 'DIRECT' if the sequential file has fixed-length records).

Sequential Access

Sequential access transfers records sequentially to or from files or I/O devices such as terminals. You can use sequential I/O with any type of supported file organization and record type.

If you select sequential access mode for files with sequential or relative organization, records are written to or read from the file starting at the beginning of the file and continuing through it, one record after another. A particular record can be retrieved only after all of the records preceding it have been read; new records can be written only at the end of the file.

Direct Access

Direct access transfers records selected by record number to and from either sequential files stored on disk with a fixed-length record type or relative organization files.

If you select direct access mode, you can determine the order in which records are read or written. Each READ or WRITE statement must include the relative record number, indicating the record to be read or written.

You can directly access a sequential disk file only if it contains fixed-length records. Because direct access uses cell numbers to find records, you can enter successive READ or WRITE statements requesting records that either precede or follow previously requested records. For example, the first of the following statements reads record 24; the second reads record 10:

```
READ (12,REC=24) I
READ (12,REC=10) J
```

Stream Access

Stream access transfers bytes from records sequentially until a record terminator is found or a specified number of bytes have been read or written. Formatted stream records are terminated with a new line character; unformatted stream data contains no record terminators. Bytes can be read from or written to a file by byte position, where the first byte of the file is position 1. An example follows:

```
OPEN (UNIT=12, ACCESS='STREAM')
READ (12) I, J, K              ! start at the first byte of the file
READ (12, POS=200) X, Y       ! then read staring at byte 200
READ (12) A, B            ! then read starting where the previous READ stopped
```

The POS= specifier on INQUIRE can be used to determine the current byte position in the file.

NOTE. The RECORDTYPE= specifier is ignored for stream access.

Limitations of Record Access by File Organization and Record Type

You can use sequential and direct access modes on sequential and relative files. However, direct access to a sequential organization file can only be done if the file resides on disk and contains fixed-length records.

The table below summarizes the types of access permitted for the various combinations of file organizations and record types.

Record Type	Sequential Access?	Direct Access?
Sequential file organization		
Fixed	Yes	Yes
Variable	Yes	No
Segmented	Yes	No

Record Type	Sequential Access?	Direct Access?
Stream	Yes	No
Stream_CR	Yes	No
Stream_LF	Yes	No
Stream_CRLF	Yes	No
Relative file organization		
Fixed	Yes	Yes

NOTE. Direct access and relative files require that the file resides on a disk device.

Record Transfer

I/O statements transfer all data as records. The amount of data that a record can contain depends on the following circumstances:

- With formatted I/O (except for fixed-length records), the number of items in the I/O statement and its associated format specifier jointly determine the amount of data to be transferred.
- With namelist and list-directed output, the items listed in the NAMELIST statement or I/O statement list (in conjunction with the NAMELIST or list-directed formatting rules) determine the amount of data to be transferred.
- With unformatted I/O (except for fixed-length records), the I/O statement alone specifies the amount of data to be transferred.
- When you specify fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding).

Typically, the data transferred by an I/O statement is read from or written to a single record. It is possible, however, for a single I/O statement to transfer data from or to more than one record, depending on the form of I/O used.

Input Record Transfer

When using advancing I/O, if an input statement specifies fewer data fields (less data) than the record contains, the remaining fields are ignored.

If an input statement specifies more data fields than the record contains, one of the following occurs:

- For formatted input using advancing I/O, if the file was opened with PAD='YES', additional fields are read as spaces. If the file is opened with PAD='NO', an error occurs (the input statement should not specify more data fields than the record contains).
- For formatted input using nonadvancing I/O (ADVANCE='NO'), an end-of-record (EOR) condition is returned. If the file was opened with PAD='YES', additional fields are read as spaces.
- For list-directed input, another record is read.
- For NAMELIST input, another record is read.
- For unformatted input, an error occurs.

Output Record Transfer

If an output statement specifies fewer data fields than the record contains (less data than required to fill a record), the following occurs:

- With fixed-length records (RECORDTYPE= 'FIXED'), all records are the same size. If the size of an I/O record being written is less than the record length (RECL), extra bytes are added (padding) in the form of spaces (for a formatted record) or zeros (for an unformatted record).
- With other record types, the fields present are written and those omitted are not written (might result in a short record).

If the output statement specifies more data than the record can contain, an error occurs, as follows:

- With formatted or unformatted output using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), an error occurs.
- With formatted or unformatted output not using fixed-length records, if the items in the output statement and its associated format specifier result in a number of bytes that exceeds the maximum record length (RECL), the Intel Fortran RTL attempts to increase the RECL value and write the longer record. To obtain the RECL value, use an INQUIRE statement.

- For list-directed output and namelist output, if the data specified exceeds the maximum record length (RECL), another record is written.

Specifying Default Pathnames and File Names

Intel Fortran provides a number of ways of specifying all or part of a file specification (directory and file name). The following list uses the Linux* pathname `/usr/proj/testdata` as an example:

- The FILE specifier in an OPEN statement typically specifies only a file name (such as testdata) or contains both a directory and file name (such as `/usr/proj/testdata`).
- The DEFAULTFILE specifier in an OPEN statement typically specifies a pathname that contains only a directory (such as `/usr/proj/`) or both a directory and file name (such as `/usr/proj/testdata`).
- If you used an implied OPEN or if the FILE specifier in an OPEN statement did not specify a file name, you can use an environment variable to specify a file name or a pathname that contains both a directory and file name.

Examples of Applying Default Pathnames and File Names

For example, for an implied OPEN of unit number 3, Intel Fortran will check the environment variable FORT3. If the environment variable FORT3 is set, its value is used. If it is not set, the system supplies the file name `fort.3`.

In the following table, assume the current directory is `/usr/smith` and the I/O uses unit 1, as in the statement READ (1,100).

OPEN FILE value	OPEN DEFAULTFILE value	FORT1 environment variable value	Resulting pathname
not specified	not specified	not specified	/usr/smith/fort.1
not specified	not specified	test.dat	/usr/smith/test.dat
not specified	not checked	/usr/tmp/t.dat	/usr/tmp/t.dat
not specified	/tmp	not specified	/tmp/fort.1
not specified	/tmp	testdata	/tmp/testdata
not specified	/usr	lib/testdata	/usr/lib/testdata

OPEN FILE value	OPEN DEFAULTFILE value	FORT1 environment variable value	Resulting pathname
file.dat	/usr/group	not checked	/usr/group/file.dat
/tmp/file.dat	not checked	not checked	/tmp/file.dat
file.dat	not specified	not checked	/usr/smith/file.dat

When the resulting file pathname begins with a tilde character (~), C-shell-style pathname substitution is used (regardless of what shell is being used), such as a top-level directory (below the root). For additional information on tilde pathname substitution, see csh(1).

Rules for Applying Default Pathnames and File Names

Intel Fortran determines the file name and the directory path based on certain rules. It determines a file name string as follows:

- If the FILE specifier is present, its value is used.
- If the FILE specifier is not present, Intel Fortran examines the corresponding environment variable. If the corresponding environment variable is set, that value is used. If the corresponding environment variable is not set, a file name in the form fort. `n` is used.

Once Intel Fortran determines the resulting file name string, it determines the directory (which optionally precedes the file name) as follows:

- If the resulting file name string contains an absolute pathname, it is used and the DEFAULTFILE specifier, environment variable, and current directory values are ignored.
- If the resulting file name string does not contain an absolute pathname, Intel Fortran examines the DEFAULTFILE specifier and current directory value: If the corresponding environment variable is set and specifies an absolute pathname, that value is used. Otherwise, the DEFAULTFILE specifier value, if present, is used. If the DEFAULTFILE specifier is not present, Intel Fortran uses the current directory as an absolute pathname.

Opening Files: OPEN Statement

To open a file, you can use a preconnected file (such as for terminal output) or can open a file with an OPEN statement. The OPEN statement lets you specify the file connection characteristics and other information.

OPEN Statement Specifiers

The OPEN statement connects a unit number with an external file and allows you to explicitly specify file attributes and run-time options using OPEN statement specifiers. Once you open a file, you should close it before opening it again unless it is a preconnected file.

If you open a unit number that was opened previously (without being closed), one of the following occurs:

- If you specify a file specification that does not match the one specified for the original open, the Intel Fortran run-time system closes the original file and then opens the second file. This resets the current record position for the second file.
- If you specify a file specification that matches the one specified for the original open, the file is reconnected without the internal equivalent of the CLOSE and OPEN. This lets you change one or more OPEN statement run-time specifiers while maintaining the record position context.

You can use the INQUIRE statement to obtain information about whether or not a file is opened by your program.

Especially when creating a new file using the OPEN statement, examine the defaults (see the description of the OPEN statement in the Intel Fortran Language Reference Manual) or explicitly specify file attributes with the appropriate OPEN statement specifiers.

Specifiers for File and Unit Information

These specifiers identify file and unit information:

- UNIT specifies the logical unit number.
- FILE (or NAME) and DEFAULTFILE specify the directory and/or file name of an external file.
- STATUS or TYPE indicates whether to create a new file, overwrite an existing file, open an existing file, or use a scratch file.
- STATUS or DISPOSE specifies the file existence status after CLOSE.

Specifiers for File and Record Characteristics

These specifiers identify file and record characteristics:

- ORGANIZATION indicates the file organization (sequential or relative).
- RECORDTYPE indicates which record type to use.

- FORM indicates whether records are formatted or unformatted.
- CARRIAGECONTROL indicates the terminal control type.
- RECL or RECORDSIZE specifies the record size.

Specifier for Special File Open Routine

USEROPEN names the routine that will open the file to establish special context that changes the effect of subsequent Intel Fortran I/O statements.

Specifiers for File Access, Processing, and Position

These specifiers identify file access, processing, and position:

- ACCESS indicates the access mode (direct, sequential, or stream).
- SHARED sets file locking for shared access. Indicates that other users can access the same file.
- NOSHARED sets file locking for exclusive access. Indicates that other users who use file locking mechanisms cannot access the same file.
- SHARE specifies shared or exclusive access; for example, SHARE='DENYNONE' or SHARE='DENYRW'.
- POSITION indicates whether to position the file at the beginning of file, before the end-of-file record, or leave it as is (unchanged).
- ACTION or READONLY indicates whether statements will be used to only read records, only write records, or both read and write records.
- MAXREC specifies the maximum record number for direct access.
- ASSOCIATEVARIABLE specifies the variable containing the next record number for direct access.
- ASYNCHRONOUS specifies whether input/output should be performed asynchronously.

Specifiers for Record Transfer Characteristics

These specifiers identify record transfer characteristics:

- BLANK indicates whether to ignore blanks in numeric fields.
- DELIM specifies the delimiter character for character constants in list-directed or namelist output.

- PAD, when reading formatted records, indicates whether padding characters should be added if the item list and format specification require more data than the record contains.
- BUFFERED indicates whether buffered or non-buffered I/O should be used.
- BLOCKSIZE specifies the block physical I/O buffer size.
- BUFFERCOUNT specifies the number of physical I/O buffers.
- CONVERT specifies the format of unformatted numeric data.

Specifiers for Error-Handling Capabilities

These specifiers are used for error handling:

- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies the integer variable to receive the error (IOSTAT) number if an error occurs.

Specifier for File Close Action

DISPOSE identifies the action to take when the file is closed.

Specifying File Locations in an OPEN Statement

You can use the FILE and DEFAULTFILE specifiers of the OPEN statement to specify the complete definition of a particular file to be opened on a logical unit. (The Language Reference Manual describes the OPEN statement in greater detail.)

For example:

```
OPEN (UNIT=4, FILE='/usr/users/smith/test.dat', STATUS='OLD')
```

The file `test.dat` in directory `/usr/users/smith` is opened on logical unit 4. No defaults are applied, because both the directory and file name were specified. The value of the FILE specifier can be a character constant, variable, or expression.

In the following interactive example, the user supplies the file name and the DEFAULTFILE specifier supplies the default values for the full pathname string. The file to be opened is located in `/usr/users/smith` and is concatenated with the file name typed by the user into the variable DOC:

```
CHARACTER(LEN=9) DOC
WRITE (6,*)  'Type file name '
READ (5,*) DOC
OPEN (UNIT=2, FILE=DOC, DEFAULTFILE='/usr/users/smith',STATUS='OLD')
```

A slash (backslash on Windows systems) is appended to the end of the default file string if it does not have one.

Obtaining File Information: INQUIRE Statement

The INQUIRE statement returns information about a file and has the following forms:

- Inquiry by unit
- Inquiry by file name
- Inquiry by output item list
- Inquiry by directory

Inquiry by Unit

An inquiry by unit is usually done for an opened (connected) file. An inquiry by unit causes the Intel Fortran RTL to check whether the specified unit is connected or not. One of the following occurs, depending on whether the unit is connected or not:

If the unit is connected:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable (if the file is named).
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.
- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If the unit is not connected:

- The OPENED specifier indicates a false value.
- The unit NUMBER specifier variable is returned as a value of `-1`.
- Any other information returned will be undefined or default values for the various specifiers.

For example, the following INQUIRE statement shows whether unit 3 has a file connected (OPENED specifier) in logical variable I_OPENED, the name (case-sensitive) in character variable I_NAME, and whether the file is opened for READ, WRITE, or READWRITE access in character variable I_ACTION:

```
INQUIRE (3, OPENED=I_OPENED, NAME=I_NAME, ACTION=I_ACTION)
```

Inquiry by File Name

An inquiry by name causes the Intel Fortran RTL to scan its list of open files for a matching file name. One of the following occurs, depending on whether a match occurs or not:

If a match occurs:

- The EXIST and OPENED specifier variables indicate a true value.
- The pathname and file name are returned in the NAME specifier variable.
- The UNIT number is returned in the NUMBER specifier variable.
- Other information requested on the previously connected file is returned.
- Default values are usually returned for the INQUIRE specifiers also associated with the OPEN statement.
- The RECL value unit for connected formatted files is always 1-byte units. For unformatted files, the RECL unit is 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

If no match occurs:

- The OPENED specifier variable indicates a false value.
- The unit NUMBER specifier variable is returned as a value of `-1`.
- The EXIST specifier variable indicates (true or false) whether the named file exists on the device or not.
- If the file does exist, the NAME specifier variable contains the pathname and file name.
- Any other information returned will be default values for the various specifiers, based on any information specified when calling INQUIRE.

The following INQUIRE statement returns whether the file named log_file is connected in logical variable I_OPEN, whether the file exists in logical variable I_EXIST, and the unit number in integer variable I_NUMBER:

```
INQUIRE (FILE='log_file', OPENED=I_OPEN, EXIST=I_EXIST, NUMBER=I_NUMBER)
```

Inquiry by Output Item List

Unlike inquiry by unit or inquiry by name, inquiry by output item list does not attempt to access any external file. It returns the length of a record for a list of variables that would be used for unformatted WRITE, READ, and REWRITE statements. The following INQUIRE statement returns the maximum record length of the variable list in variable I_RECLENGTH. This variable is then used to specify the RECL value in the OPEN statement:

```
INQUIRE (IOLENGTH=I_RECLENGTH) A, B, H
OPEN (FILE='test.dat', FORM='UNFORMATTED', RECL=I_RECLENGTH, UNIT=9)
```

For an unformatted file, the IOLENGTH value is returned using 4-byte units, unless you specify the `-assume byterecl` option to request 1-byte units.

Inquiry by Directory

An inquiry by directory verifies that a directory exists.

If the directory exists:

- The EXIST specifier variable indicates a true value.
- The full directory pathname is returned in the DIRSPEC specifier variable.

If the directory does not exist:

- The EXIST specified variable indicates a false value.
- The value of the DIRSPEC specifier variable is unchanged.

For example, the following INQUIRE statement returns the full directory pathname:

```
LOGICAL        ::L_EXISTS
CHARACTER (255)::C_DIRSPEC
INQUIRE (DIRECTORY=".", DIRSPEC=C_DIRSPEC, EXIST=L_EXISTS)
```

The following INQUIRE statement verifies that a directory does not exist:

```
INQUIRE (DIRECTORY="I-DO-NOT-EXIST",
EXIST=L_EXISTS)
```

Closing Files: CLOSE Statement

Usually, any external file opened should be closed by the same program before it completes. The CLOSE statement disconnects the unit and its external file. You must specify the unit number (UNIT specifier) to be closed.

You can also specify:

- Whether the file should be deleted or kept (STATUS specifier)
- Error handling information (ERR and IOSTAT specifiers)

To delete a file when closing it:

- In the OPEN statement, specify the ACTION keyword (such as ACTION='READ'). Avoid using the READONLY keyword, because a file opened using the READONLY keyword cannot be deleted when it is closed.
- In the CLOSE statement, specify the keyword STATUS='DELETE'.

If you opened an external file and did an inquire by unit, but do not like the default value for the ACCESS specifier, you can close the file and then reopen it, explicitly specifying the ACCESS desired.

There usually is no need to close preconnected units. Internal files are neither opened nor closed.

Record I/O Statement Specifiers

After you open a file or use a preconnected file, you can use the following statements:

- READ, WRITE, ACCEPT, and PRINT to perform record I/O.
- BACKSPACE, ENDFILE, and REWIND to set record position within the file.
- DELETE, REWRITE, TYPE, and FIND to perform various operations.

The record I/O statement must use the appropriate record I/O form (formatted, list-directed, namelist, or unformatted).

You can use the following specifiers with the READ and WRITE record I/O statements:

- UNIT specifies the unit number to or from which input or output will occur.
- END specifies a label to branch to if end-of-file occurs; only applies to input statements on sequential files.
- ERR specifies a label to branch to if an error occurs.
- IOSTAT specifies an integer variable to contain the error number if an error occurs.
- FMT specifies a label of a FORMAT statement or character data specifying a FORMAT.
- NML specifies the name of a NAMELIST.
- REC specifies a record number for direct access.

When using nonadvancing I/O, use the ADVANCE, EOR, and SIZE specifiers.

When using the REWRITE statement, you can use the UNIT, FMT, ERR, and IOSTAT specifiers.

File Sharing on Linux* OS and Mac OS* X Systems

Depending on the value specified by the ACTION (or READONLY) specifier in the OPEN statement, the file will be opened by your program for reading, writing, or both reading and writing records. This simply checks that the program itself executes the type of statements intended.

File locking mechanisms allow users to enable or restrict access to a particular file when that file is being accessed by another process.

Intel® Fortran file locking features provide three file access modes:

- Implicit Shared mode, which occurs when no mode is specified. This is also called No Locking.
- Explicit Shared mode, when all cooperating processes have access to a file. This mode is set in the OPEN statement by the SHARED specifier or the SHARE='DENYNONE' specifier.
- Exclusive mode, when only one process has access to a file. This mode is set in the OPEN statement by the NOSHARED specifier or the SHARE='DENYRW' specifier.

The file locking mechanism looks for explicit setting of the corresponding specifier in the OPEN statement. Otherwise, the Fortran run time does not perform any setting or checking for file locking and the process can access the file regardless of the fact that other processes have already opened or locked the file.

Example 1: Implicit Shared Mode (No Locking)

Process 1 opens the file without a specifier, resulting in no locking.

Process 2 now tries to open the file:

- It gains access regardless of the mode it is using.

Example 2: Explicit Shared Mode

Process 1 opens the file with Explicit Shared mode.

Process 2 now tries to open the file:

- If process 2 opens the file with Explicit Shared mode or Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Exclusive mode, it receives an error.

Example 3: Exclusive Mode

Process 1 opens the file with Exclusive mode.

Process 2 now tries to open the file:

- If process 2 opens the file with Implicit Shared (No Locking) mode, it gets access to the file.
- If process 2 opens the file with Explicit Shared or Exclusive mode, it receives an error.

The Fortran runtime does not coordinate file entry updates during cooperative access. The user needs to coordinate access times among cooperating processes to handle the possibility of simultaneous WRITE and REWRITE statements on the same record positions.

Specifying the Initial Record Position

When you open a disk file, you can use the OPEN statement POSITION specifier to request one of the following initial record positions within the file:

- The initial position before the first record (POSITION='REWIND'). A sequential access READ or WRITE statement will read or write the first record in the file.
- A point beyond the last record in the file (POSITION='APPEND'), just before the end-of-file record, if one exists. For a new file, this is the initial position before the first record (same as 'REWIND'). You might specify 'APPEND' before you write records to an existing sequential file using sequential access.
- The current position (POSITION='ASIS') . This is usually used only to maintain the current record position when reconnecting a file. The second OPEN specifies the same unit number and specifies the same file name (or omits it), which leaves the file open, retaining the current record position. However, if the second OPEN specifies a different file name for the same unit number, the current file will be closed and the different file will be opened.

The following I/O statements allow you to change the current record position:

- REWIND sets the record position to the initial position before the first record. A sequential access READ or WRITE statement would read or write the first record in the file.
- BACKSPACE sets the record position to the previous record in a file. Using sequential access, if you wrote record 5, issued a BACKSPACE to that unit, and then read from that unit, you would read record 5.
- ENDFILE writes an end-of-file marker. This is typically done after writing records using sequential access just before you close the file.

Unless you use nonadvancing I/O, reading and writing records usually advances the current record position by one record. More than one record might be transferred using a single record I/O statement.

Advancing and Nonadvancing Record I/O

After you open a file, if you omit the ADVANCE specifier (or specify ADVANCE= 'YES') in READ and WRITE statements, advancing I/O (normal `Fortran I/O`) will be used for record access. When using advancing I/O:

- Record I/O statements transfer one entire record (or multiple records).
- Record I/O statements advance the current record position to a position before the next record.

You can request nonadvancing I/O for the file by specifying the ADVANCE= 'NO' specifier in a READ and WRITE statement. You can use nonadvancing I/O only for sequential access to external files using formatted I/O (not list-directed or namelist).

When you use nonadvancing I/O, the current record position does not change, and part of the record might be transferred, unlike advancing I/O where one or more entire records are always transferred.

You can alternate between advancing and nonadvancing I/O by specifying different values for the ADVANCE specifier ('YES' and 'NO') in the READ and WRITE record I/O statements.

When reading records with either advancing or nonadvancing I/O, you can use the `END specifier` to branch to a specified label when the end of the file is read.

Because nonadvancing I/O might not read an entire record, it also supports an `EOR specifier` to branch to a specified label when the end of the record is read. If you omit the EOR and the IOSTAT specifiers when using nonadvancing I/O, an error results when the end-of-record is read.

When using nonadvancing input, you can use the SIZE specifier to return the number of characters read. For example, in the following READ statement, SIZE=X (where variable X is an integer) returns the number of characters read in X and an end-of-record condition causes a branch to label 700:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

User-Supplied OPEN Procedures: USEROPEN Specifier

You can use the USEROPEN specifier in an Intel Fortran OPEN statement to pass control to a routine that directly opens a file. The called routine can use system calls or library routines to open the file and establish special context that changes the effect of subsequent Intel Fortran I/O statements.

The Intel Fortran RTL I/O support routines call the USEROPEN function in place of the system calls usually used when the file is first opened for I/O. The USEROPEN specifier in an OPEN statement specifies the name of a function to receive control. The called function must open the file (or pipe) and return the file descriptor of the file when it returns control to the RTL.

When opening the file, the called function usually specifies options different from those provided by a normal OPEN statement.

You can obtain the file descriptor from the Intel Fortran RTL for a specific unit number with the getfd routine.

Although the called function can be written in other languages (such as Fortran), C is usually the best choice for making system calls, such as open or create.

Syntax and Behavior of the USEROPEN Specifier

The USEROPEN specifier for the OPEN statement has the form:

```
USEROPEN = function-name
```

function-name represents the name of an external function. In the calling program, the function must be declared in an EXTERNAL statement. For example, the following Intel Fortran code might be used to call the USEROPEN procedure UOPEN (known to the linker as uopen_):

```
EXTERNAL  UOPEN
INTEGER   UOPEN
.
.
.
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW', USEROPEN=UOPEN)
```

During the OPEN statement, the uopen_function receives control. The function opens the file, may perform other operations, and subsequently returns control (with the file descriptor) to the RTL.

If the USEROPEN function is written in C, declare it as a C function that returns a 4-byte integer (int) result to contain the file descriptor. For example:

```
int   uopen_ (          (1)
char  *file_name,       (2)
int   *open_flags,      (3)
int   *create_mode,     (4)
int   *lun,             (5)
int   file_length);     (6)
```

The function definition and the arguments passed from the Intel Fortran RTL are as follows:

1. The function must be declared as a 4-byte integer (int).
2. The first argument is the pathname (includes the file name) to be opened.

3. The open flags are described in the header file `/usr/include/sys/file.h` or open(2).
4. The create mode (protection needed when creating a Linux* OS-style file) is described in open(2).
5. The fourth argument is the logical unit number.
6. The fifth (last) argument is the pathname length (hidden length argument of the pathname).

Of the arguments, the open system call (see open(2)) requires the passed pathname, the open flags (that define the type access needed, whether the file exists, and so on), and the create mode. The logical unit number specified in the OPEN statement is passed in case the USEROPEN function needs it. The hidden length of the pathname is also passed.

When creating a new file, the create system call might be used in place of open (see create(2)). You can usually use other appropriate system calls or library routines within the USEROPEN function.

In most cases, the USEROPEN function modifies the open flags argument passed by the Intel Fortran RTL or uses a new value before the open (or create) system call. After the function opens the file, it must return control to the RTL.

If the USEROPEN function is written in Fortran, declare it as a FUNCTION with an INTEGER (KIND=4) result, perhaps with an interface block. In any case, the called function must return the file descriptor as a 4-byte integer to the RTL.

If your application requires that you use C to perform the file open and close, as well as all record operations, call the appropriate C procedure from the Intel Fortran program without using the Fortran OPEN statement.

Restrictions of Called USEROPEN Functions

The Intel Fortran RTL uses exactly one file descriptor per logical unit, which must be returned by the called function. Because of this, only certain system calls or library routines can be used to open the file.

System calls and library routines that do not return a file descriptor include mknod (see mknod(2)) and fopen (see fopen(3)). For example, the fopen routine returns a file pointer instead of a file descriptor.

Example USEROPEN Program and Function

The following Intel Fortran code calls the USEROPEN function named UOPEN:

```
EXTERNAL  UOPEN
INTEGER   UOPEN
.
.
.
OPEN (UNIT=1,FILE='ex1.dat',STATUS='NEW',USEROPEN=UOPEN,
ERR=9,IOSTAT=errnum)
```

If UOPEN is a Fortran function, its name is decorated appropriately for Fortran.

Likewise, if UOPEN is a C function, its name is decorated appropriately for C, as long as the following line is included in the above code:

```
!DEC$ATTRIBUTES C::UOPEN
```

Compiling and Linking the C and Intel Fortran Programs

Use the icc or icl command to compile the called `uopen` C function `uopen.c` and the `ifort` command to compile the Intel Fortran calling program `ex1.f`. The same ifort command also links both object files by using the appropriate libraries:

```
icc -c uopen.c (Linux* OS)
icl -c uopen.c (Windows* OS)
ifort ex1.f uopen.o
```

Example Source Code

```
      program UserOpenSample
!DEC$ FREEFORM
      IMPLICIT NONE
      EXTERNAL UOPEN
      INTEGER(4) UOPEN
      CHARACTER*10 :: FileName="UOPEN.DAT"
      INTEGER*4 :: IOS
      Character*255 :: InqFullName
      Character*100 :: InqFileName
      Integer :: InqLun
      Character*30 :: WriteOutBuffer="Write One Record to the File. "
      Character*30 :: ReadInBuffer  ="??????????????????????????????"
110   FORMAT( X,A, ": Created (iostat=",I0,")")
115   FORMAT( X,A, ": Creation Failed (iostat=",I0,")")
120   FORMAT( X,A, ": ERROR: INQUIRE Returned Wrong FileName")
130   FORMAT( X,A, ": ERROR: ReadIn and WriteOut Buffers Do Not Match")
190   FORMAT( X,A, ": Completed.")
      WRITE(*,'(X,"Test the USEROPEN Facility of Open")')
      OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='REPLACE',USEROPEN=UOPEN, &
```

```
            IOSTAT=ios, ACTION='READWRITE')

!     When the OPEN statement is executed,
!     the UOPEN function receives control.
!     The function opens the file by calling CreateFile( ),
!     performs whatever operations were specified, and subsequently
!     returns control (with the handle returned by CreateFile( ))
!     to the calling Fortran program.
      IF (IOS .EQ. 0)  THEN
         WRITE(*,110) TRIM(FileName), IOS
         INQUIRE(10, NAME=InqFullName)
         CALL ParseForFileName(InqFullName,InqFileName)
         IF (InqFileName .NE. FileName) THEN
            WRITE(*,120) TRIM(FileName)
         END IF
      ELSE
         WRITE(*,115) TRIM(FileName), IOS
         GOTO 9999
      END IF
      WRITE(10,*) WriteOutBuffer
      REWIND(10)
      READ(10,*)
 ReadInBuffer
      IF (ReadinBuffer .NE. WriteOutbuffer) THEN
         WRITE(*,130) TRIM(FileName)
      END IF
      CLOSE(10, DISPOSE='DELETE')
      WRITE(*,190) TRIM(FileName)
      WRITE(*,'(X,"Test of USEROPEN Completed")')
9999  CONTINUE
      END
!DEC$ IF DEFINED(_WIN32)
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! Here is the UOPEN function for WIN32:
!
! The UOPEN function is declared to use the cdecl calling convention,
! so it matches the Fortran rtl declaration of a useropen routine.
!
! The following function definition and arguments are passed from the Intel
! Fortran Run-time Library to the function named in USEROPEN:
!
! The first 7 arguments correspond to the CreateFile( ) api arguments.
! The value of these arguments is set according the caller's OPEN( )
! arguments:
!
! FILENAME
!      Is the address of a null terminated character string that
!      is the name of the file.
! DESIRED_ACCESS
!      Is the desired access (read-write) mode passed by reference.
! SHARE_MODE
```

```
!      Is the file sharing mode passed by reference.
! A_NULL
!      Is always null. The Fortran runtime library always passes a NULL
!      for the pointer to a SECURITY_ATTRIBUTES structure in its
!      CreateFile( ) call.
! CREATE_DISP
!      Is the creation disposition specifying what action to take on files
!      that exist, and what action to take on files
!      that do not exist. It is passed by reference.
! FLAGS_ATTR
!      Specifies the file attributes and flags for the file. It is passed
!      by reference.
! B_NULL
!      Is always null. The Fortran runtime library always passes a NULL
!      for the handle to a template file in it's CreateFile( ) call.
! The last 2 arguments are the Fortran unit number and length of the
! file name:
! UNIT
!      Is the Fortran unit number on which this OPEN is being done. It is
!      passed by reference.
! FLEN
!      Is the length of the file name, not counting the terminating null,
!      and passed by value.
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      INTEGER(4) FUNCTION UOPEN( FILENAME,       &
                                 DESIRED_ACCESS, &
                                 SHARE_MODE, &
                                 A_NULL, &
                                 CREATE_DISP, &
                                 FLAGS_ATTR, &
                                 B_NULL, &
                                 UNIT, &
                                 FLEN )
      !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'UOPEN' :: UOPEN
      !DEC$ATTRIBUTES REFERENCE :: FILENAME
      !DEC$ATTRIBUTES REFERENCE :: DESIRED_ACCESS
      !DEC$ATTRIBUTES REFERENCE :: SHARE_MODE
      !DEC$ATTRIBUTES REFERENCE :: CREATE_DISP
      !DEC$ATTRIBUTES REFERENCE :: FLAGS_ATTR
      !DEC$ATTRIBUTES REFERENCE :: UNIT
      USE KERNEL32
      IMPLICIT NONE
      INTEGER*4 DESIRED_ACCESS
      INTEGER*4 SHARE_MODE
      INTEGER*4 A_NULL
      INTEGER*4 CREATE_DISP
      INTEGER*4 FLAGS_ATTR
      INTEGER*4 B_NULL
      INTEGER*4 UNIT
      INTEGER*4 FLEN
      CHARACTER*(FLEN) FILENAME
```

```
      INTEGER(4) ISTAT
      TYPE(T_SECURITY_ATTRIBUTES), POINTER :: NULL_SEC_ATTR
140   FORMAT( X, "ERROR: USEROPEN Passed Wrong Unit Number",I)
!     Sanity check
      IF (UNIT .NE. 10) THEN
         WRITE(*,140) UNIT
      END IF
      !! WRITE(*,*) "FILENAME=",FILENAME !! prints the full path of the filename
! Set the FILE_FLAG_WRITE_THROUGH bit in the flag attributes to CreateFile( )
! (for whatever reason)
!     FLAGS_ATTR = FLAGS_ATTR + FILE_FLAG_WRITE_THROUGH
! Do the CreateFile( ) call and return the status to the Fortran rtl
      ISTAT = CreateFile( FILENAME,       &
                          DESIRED_ACCESS, &
                          SHARE_MODE, &
                          NULL_SEC_ATTR, &
                          CREATE_DISP, &
                          FLAGS_ATTR, &
                          0 )
      if (ISTAT == INVALID_HANDLE_VALUE) then
          write(*,*) "Could not open file (error ", GetLastError(),")"
      endif
      UOPEN = ISTAT
      RETURN
      END
!DEC$ ELSE ! LINUX OS or MAC OS X
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! Here is the UOPEN function for Linux OS/Mac OS X:
!
! The UOPEN function is declared to use the cdecl calling convention,
! so it matches the Fortran rtl declaration of a useropen routine.
!
! The following function definition and arguments are passed from the
! Intel Fortran Run-time Library to the function named in USEROPEN:
!
! FILENAME
!      Is the address of a null terminated character string that
!      is the name of the file.
! OPEN_FLAGS
!      read-write flags (see file.h or open(2)).
! CREATE_MODE
!      set if new file (to be created).
! UNIT
!      Is the Fortran unit number on which this OPEN is being done. It is
!      passed by reference.
! FLEN
!      Is the length of the file name, not counting the terminating null,
!      and passed by value.
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      INTEGER FUNCTION UOPEN( FILENAME,       &
                              OPEN_FLAGS, &
```

```
                              CREATE_MODE, &
                              UNIT, &
                              FLEN )
      !DEC$ATTRIBUTES C, DECORATE, ALIAS:'uopen' :: UOPEN
      !DEC$ATTRIBUTES REFERENCE :: FILENAME
      !DEC$ATTRIBUTES REFERENCE :: OPEN_FLAGS
      !DEC$ATTRIBUTES REFERENCE :: CREATE_MODE
      !DEC$ATTRIBUTES REFERENCE :: UNIT
      IMPLICIT NONE
      INTEGER*4 OPEN_FLAGS
      INTEGER*4 CREATE_MODE
      INTEGER*4 UNIT
      INTEGER*4 FLEN
      CHARACTER*(FLEN) FILENAME
      INTEGER*4 ISTAT
      !DEC$ATTRIBUTES C, DECORATE, ALIAS:'open'  :: OPEN
      external OPEN
      integer*4 OPEN
140   FORMAT( X, "ERROR: USEROPEN Passed Wrong Unit Number",I)
!     Sanity check
      IF (UNIT .NE. 10) THEN
         WRITE(*,140) UNIT
      END IF
!     Call the system OPEN routine
      ISTAT = OPEN ( %ref(FILENAME),     &
                     OPEN_FLAGS, &
                     CREATE_MODE )
      UOPEN = ISTAT
      RETURN
      END
!DEC$ ENDIF ! End of UOPEN Function
!---------------------------------------------------------------
! SUBROUTINE: ParseForFileName
!             Takes a full pathname and retuns the filename
!             with its extension.
!---------------------------------------------------------------
      SUBROUTINE ParseForFileName(FullName,FileName)
      Character*255 :: FullName
      Character*100 :: FileName
      Integer       :: P
!DEC$ IF DEFINED(_WIN32)
      P = INDEX(FullName,'\',.TRUE.)
      FileName = FullName(P+1:)
!DEC$ ELSE ! LINUX OS/MAC OS X
      P = INDEX(FullName,'/',.TRUE.)
      FileName = FullName(P+1:)
!DEC$ ENDIF
      END
```

Microsoft Fortran PowerStation Compatible Files

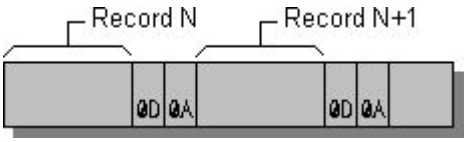
When using the `-fpscomp` (Linux OS and Mac OS X) or `/fpscomp` (Windows OS) options for Microsoft* Fortran PowerStation compatibility, the following types of files are possible:

- Formatted Sequential
- Formatted Direct
- Unformatted Sequential
- Unformatted Direct
- Binary Sequential
- Binary Direct

Formatted Sequential Files

A formatted sequential file is a series of formatted records written sequentially and read in the order in which they appear in the file. Records can vary in length and can be empty. They are separated by carriage return (0D) and line feed (0A) characters as shown in the following figure.

Figure 12: Formatted Records in a Formatted Sequential File

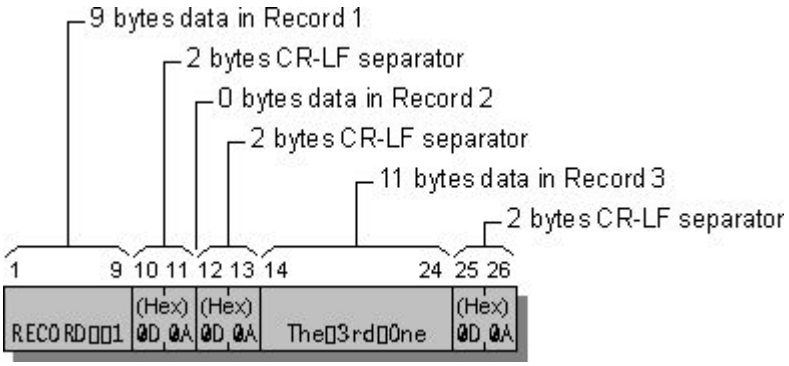


An example of a program writing three records to a formatted sequential file is given below. The resulting file is shown in the following figure.

```
   OPEN (3, FILE='FSEQ')
! FSEQ is a formatted sequential file by default.
  WRITE (3, '(A, I3)') 'RECORD', 1
  WRITE (3, '()')
```

```
WRITE (3, '(A11)') 'The 3rd One'
CLOSE (3)
END
```

Figure 13: Formatted Sequential File



Formatted Direct Files

In a formatted direct file, all of the records are the same length and can be written or read in any order. The record size is specified with the RECL option in an `OPEN` statement and should be equal to or greater than the number of bytes in the longest record.

The carriage return (CR) and line feed (LF) characters are record separators and are not included in the RECL value. Once a direct-access record has been written, you cannot delete it, but you can rewrite it.

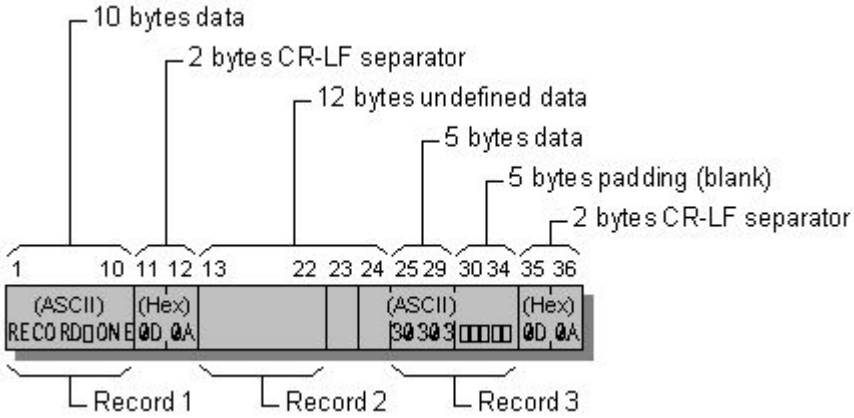
During output to a formatted direct file, if data does not completely fill a record, the compiler pads the remaining portion of the record with blank spaces. The blanks ensure that the file contains only completely filled records, all of the same length. During input, the compiler by default also adds filler bytes (blanks) to the input record if the input list and format require more data than the record contains.

You can override the default blank padding on input by setting PAD='NO' in the `OPEN` statement for the file. If PAD='NO', the input record must contain the amount of data indicated by the input list and format specification. Otherwise, an error occurs. PAD='NO' has no effect on output.

An example of a program writing two records, record one and record three, to a formatted direct file is given below. The result is shown in the following figure.

```
OPEN (3,FILE='FDIR', FORM='FORMATTED', ACCESS='DIRECT',RECL=10)
WRITE (3, '(A10)', REC=1) 'RECORD ONE'
WRITE (3, '(I5)', REC=3) 30303
CLOSE (3)
END
```

Figure 14: Formatted Direct File



Unformatted Sequential Files

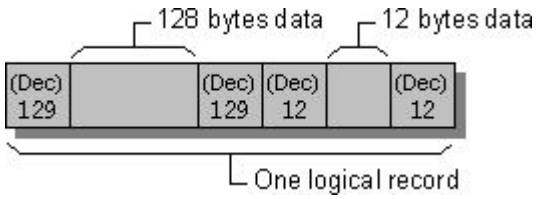
Unformatted sequential files are organized slightly differently on different platforms. This section describes unformatted sequential files created by Intel Fortran when the `-fpscomp` (Linux OS and Mac OS X) or `/fpscomp` (Windows OS) option is specified.

The records in an unformatted sequential file can vary in length. Unformatted sequential files are organized in chunks of 130 bytes or less called *`physical blocks`*. Each physical block consists of the data you send to the file (up to 128 bytes) plus two 1-byte "length bytes" inserted by the compiler. The length bytes indicate where each record begins and ends.

A *`logical record`* refers to an unformatted record that contains one or more physical blocks. (See the following figure.) Logical records can be as big as you want; the compiler will use as many physical blocks as necessary.

When you create a logical record consisting of more than one physical block, the compiler sets the length byte to 129 to indicate that the data in the current physical block continues on into the next physical block. For example, if you write 140 bytes of data, the logical record has the structure shown in the following figure.

Figure 15: Logical Record in Unformatted Sequential File



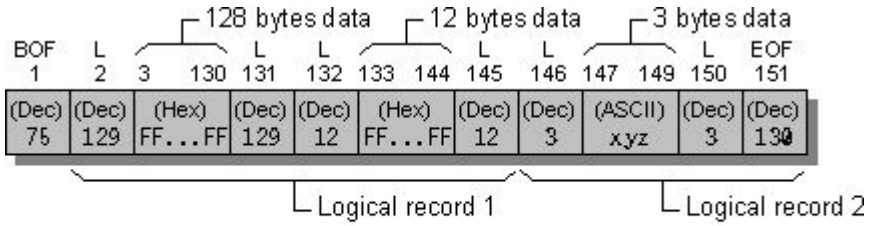
The first and last bytes in an unformatted sequential file are reserved; the first contains a value of 75, and the last holds a value of 130. Fortran uses these bytes for error checking and end-of-file references.

The following program creates the unformatted sequential file shown in the following figure:

```
!   Note: The file is sequential by default
!         -1 is FF FF FF FF hexadecimal.
!
  CHARACTER xyz(3)
  INTEGER(4) idata(35)
  DATA      idata /35 * -1/, xyz /'x', 'y', 'z'/
!
! Open the file and write out a 140-byte record:
! 128 bytes (block) + 12 bytes = 140 for IDATA, then 3 bytes for XYZ.
  OPEN (3, FILE='UFSEQ',FORM='UNFORMATTED')
  WRITE (3) idata
```

```
WRITE (3) xyz
CLOSE (3)
END
```

Figure 16: Unformatted Sequential File



BOF Beginning-of-file byte (75 decimal)
L Physical-block-length byte (0 <= L <= 129)
EOF End-of-file byte (130 decimal)

Unformatted Direct Files

An unformatted direct file is a series of unformatted records. You can write or read the records in any order you choose. All records have the same length, given by the RECL specifier in an `OPEN` statement. No delimiting bytes separate records or otherwise indicate record structure.

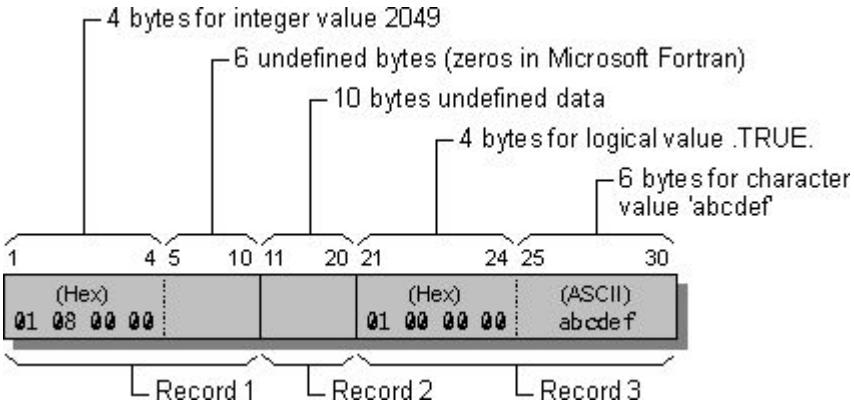
You can write a partial record to an unformatted direct file. Intel Fortran pads these records to the fixed record length with ASCII NULL characters. Unwritten records in the file contain undefined data.

The following program creates the sample unformatted direct file shown in the following figure:

```
 OPEN (3, FILE='UFDIR', RECL=10,&
  & FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
```

```
  WRITE (3, REC=1) 2049
  CLOSE (3)
  END
```

Figure 17: Unformatted Direct File



Binary Sequential Files

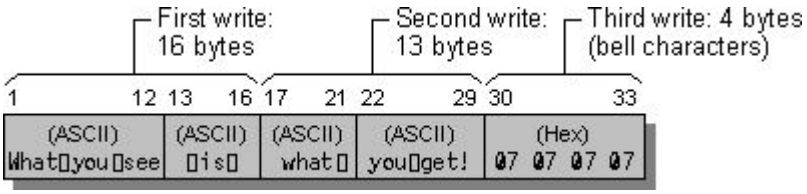
A binary sequential file is a series of values written and read in the same order and stored as binary numbers. No record boundaries exist, and no special bytes indicate file structure. Data is read and written without changes in form or length. For any I/O data item, the sequence of bytes in memory is the sequence of bytes in the file.

The next program creates the binary sequential file shown in the following figure:

```
!  NOTE: 07 is the bell character
!     Sequential is assumed by default.
!
  INTEGER(1) bells(4)
  CHARACTER(4) wys(3)
  CHARACTER(4) cvar
  DATA bells /4*7/
  DATA cvar /' is '/,wys /'What',' you',' see'/
  OPEN (3, FILE='BSEQ',FORM='BINARY')
  WRITE (3) wys, cvar
  WRITE (3) 'what ', 'you get!'
```

```
WRITE (3) bells
CLOSE (3)
END
```

Figure 18: Binary Sequential File



Binary Direct Files

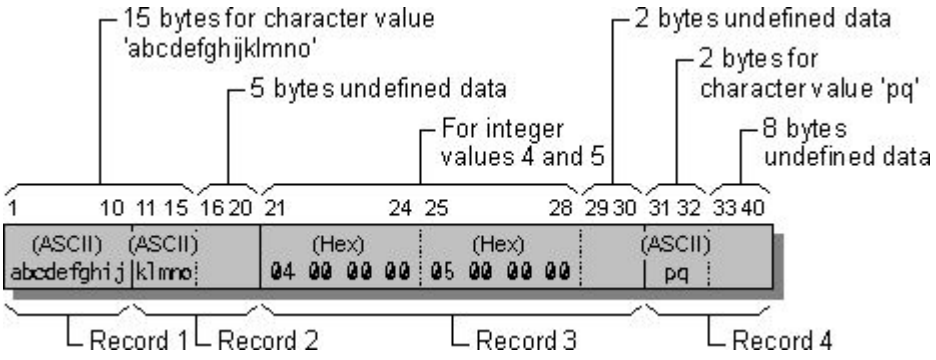
A binary direct file stores records as a series of binary numbers, accessible in any order. Each record in the file has the same length, as specified by the RECL argument to the `OPEN` statement. You can write partial records to binary direct files; any unused portion of the record will contain undefined data.

A single read or write operation can transfer more data than a record contains by continuing the operation into the next records of the file. Performing such an operation on an unformatted direct file would cause an error. Valid I/O operations for unformatted direct files produce identical results when they are performed on binary direct files, provided the operations do not depend on zero padding in partial records.

The following program creates the binary direct file shown in the following figure:

```
 OPEN (3, FILE='BDIR',RECL=10,FORM='BINARY',ACCESS='DIRECT')
WRITE (3, REC=1) 'abcdefghijklmno'
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)  END
```

Figure 19: Binary Direct File



Using Asynchronous I/O

For external files, you can specify that I/O should be asynchronous. By doing this, you allow other statements to execute while an I/O statement is executing.

NOTE. In order to execute a program that uses asynchronous I/O on Linux* OS or Mac OS* X systems, you must explicitly include one of the following compiler command line options when you compile and link your program:

- `-threads`
- `-reentrancy threaded`
- `-openmp`

On Windows* OS systems, no extra options are needed to execute a program that uses asynchronous I/O.

Using the ASYNCHRONOUS Specifier

Asynchronous I/O is supported for all READ and WRITE operations to external files. However, if you specify asynchronous I/O, you cannot use variable format expressions in formatted I/O operations.

To allow asynchronous I/O for a file, first specify ASYNCHRONOUS='YES' in its OPEN statement, then do the same for each READ or WRITE statement that you want to execute in this manner.

Execution of an asynchronous I/O statement initiates a "pending" I/O operation, which can be terminated in the following ways:

- by an explicit WAIT (initno) statement, which performs a wait operation for the specified pending asynchronous data transfer operation
- by a CLOSE statement for the file
- by a file-positioning statement such as REWIND or BACKSPACE
- by an INQUIRE statement for the file

Use the WAIT statement to ensure that the objects used in the asynchronous data transfer statements are not prematurely deallocated. (This is especially important for local stack objects and allocatable objects which may be deallocated before completion of the pending operation.) If you do not specify the wait operation, the program may terminate with an Access violation error. The following example shows use of the WAIT statement:

```
module mod
   real, allocatable :: X(:)
end module mod
subroutine sbr()
use mod
integer :: Y(500)
   !X and Y initialization
   allocate (X(500))
   call foo1(X, Y)
   !asynchronous writing
   open(1, asynchronous='yes')
   write(1, asynchronous='yes') X, Y
   !some computation
   call foo2()
   !wait operation
   wait(1)
   !X deallocation
   deallocate(X)
   !stack allocated object Y will be deallocated when the routine returns
end subroutine sbr
```

You can use the INQUIRE statement with the keyword of ASYNCHRONOUS (ASYNCHRONOUS=specifier) to determine whether asynchronous I/O is allowed. If it is allowed, a value of YES is returned.

Additionally, you can use the INQUIRE statement with the keyword of PENDING (PENDING= specifier) to determine whether previously pending asynchronous data transfers are complete.

If an ID= specifier appears and the specified data transfer operation is complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs a wait operation for the specified data transfer.

If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit are complete, the variable specified by PENDING is assigned the value False and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.

Otherwise, the variable specified by PENDING is assigned the value True and no wait operations are performed. Previously pending data transfers remain pending.

Using the ASYNCHRONOUS Attribute

A data attribute called ASYNCHRONOUS specifies that a variable may be subject to asynchronous input/output. Assigning this attribute to a variable allows certain optimizations to occur.

For more information, see the following topics in the Language Reference:

Asynchronous Specifier

Open: ASYNCHRONOUS Specifier

INQUIRE: ASYNCHRONOUS Specifier

ASYNCHRONOUS Statement and Attributes

Structuring Your Program 12

Structuring Your Program Overview

There are several ways to organize your projects and the applications that you build with Intel® Fortran. This section introduces several of these options and offers suggestions for when you might want to use them.

See Also

- Structuring Your Program
- Creating Fortran Executables
- Using Module (.mod) Files
- Using Include Files
- Advantages of Internal Procedures
- Storing Object Code in Static Libraries
- Storing Routines in Shareable Libraries

Creating Fortran Executables

The simplest way to build an application is to compile all of your Intel® Fortran source files and then link the resulting object files into a single executable file. You can build single-file executables using the ifort command from the command line. For Windows* OS, you can also use the visual development environment.

The executable file you build with this method contains all of the code needed to execute the program, including the run-time library. Because the program resides in a single file, it is easy to copy or install. However, the project contains all of the source and object files for the routines that you used to build the application. If you need to use some of these routines in other projects, you must link all of them again.

Exceptions to this are as follows:

- If you are using shared libraries, all code will not be contained in the executable file.
- On Mac OS* X, the object files contain debug information and would need to be copied along with the executable.

Using Module (.mod) Files

One way to reduce potential confusion when you use the same source code in several projects is to organize the routines into modules. A module (.mod file) is a type of program unit that contains specifications of such entities as data objects, parameters, structures, procedures, and operators. These precompiled specifications and definitions can be used by one or more program units. Partial or complete access to the module entities is provided by the a program's USE statement. Typical applications of modules are the specification of global data or the specification of a derived type and its associated operations.

Modules are excellent ways to organize programs. You can set up separate modules for:

- Commonly used routines
- Data definitions specific to certain operating systems
- System-dependent language extensions

Some programs require modules located in multiple directories. You can use the `-I` (Linux* OS and Mac OS* X) or `/I` compiler (Windows* OS) option when you compile the program to specify the location of the `.mod` files that should be included in the program.

You can use the `-module` path (Linux OS and Mac OS X) or `/module:path` (Windows OS) option to specify the directory in which to create the module files. If you don't use this option, module files are created in the current directory.

Directories are searched for `.mod` files in this order:

1. Directory of the source file that contains the USE statement
2. Directories specified by the `-module path` (Linux OS and Mac OS X) or `/module:path` (Windows OS) option
3. Current working directory
4. Directories specified by the `-Idir` (Linux OS and Mac OS X) or `/include` (Windows OS) option
5. Directories specified with the `FPATH` (Linux OS and Mac OS X) or `INCLUDE` (Windows OS) environment variable
6. Standard system directories

You need to make sure that the module files are created before they are referenced by another program or subprogram.

Compiling Programs with Modules

If a file being compiled has one or more modules defined in it, the compiler generates one or more `.mod` files.

For example, a file `a.f90` contains modules defined as follows:

```
module test
integer:: a
contains
  subroutine f()
  end subroutine
end module test
module payroll
.
.
.
end module payroll
```

This compiler command:

```
ifort -ca.f90
```

generates the following files:

- `test.mod`
- `payroll.mod`
- `a.o` (Linux OS and Mac OS X) or `a.obj` (Windows OS)

The `.mod` files contain the necessary information regarding the modules that have been defined in the program `a.f90`.

The following example uses the program program mod_def.f90 which contains a module defined as follows:

```
file: mod_def.f90
module definedmod
.
.
.
end module
```

Compile the program as follows:

```
ifort -c mod_def.f90
```

This produces the object files `mod_def.o` (Linux OS and Mac OS X) or `mod_def.obj` (Windows OS) and also the .mod file `definedmod.mod`, all in the current directory.

If you need to use the .mod file in another directory, do the following:

```
file: use_mod_def.f90
program usemod
use definedmod
.
.
.
end program
```

To compile the above program, use the -I (Linux OS) or /I (Windows OS) option to specify the path to search and locate the definedmod.mod file.

Using Include Files

Include files are brought into a program with the #include preprocessor directive or a Fortran INCLUDE statement.

Directories are searched for include files in this order:

1. Directory of the source file that contains the include
2. Current working directory
3. Directories specified by the -I (Linux OS and Mac OS X) or /I (Windows OS) option
4. Directory specified by the -isystem option (Linux OS and Mac OS X systems only)
5. Directories specified with the FPATH (Linux OS and Mac OS X) or INCLUDE (Windows OS) environment variable
6. Standard system directories

The locations of directories to be searched are known as the include file path. More than one directory can be specified in the include file path.

Specifying and Removing an Include File Path

You can use the -I (Linux OS and Mac OS X) or /I (Windows OS) option to indicate the location of include files (and also module files).

To prevent the compiler from searching the default path specified by the FPATH or the INCLUDE environment variable, use the -X or /noinclude option.

You can specify these options in the configuration file, ifort.cfg, or on the command line.

For example, to direct the compiler to search a specified path instead of the default path, use the following command line:

```
ifort -X -I/alt/include newmain.f (Linux OS and Mac OS X)
ifort /noinclude /IC:/Project2/include newmain.f (Windows OS)
```

For more information, see the following topic:

- I compiler option

Advantages of Internal Procedures

Functions or subroutines that are used in only one program can be organized as internal procedures, following the `CONTAINS` statement of a program or module.

Internal procedures have the advantage of host association, that is, variables declared and used in the main program are also available to any internal procedure it may contain. For more information on procedures and host association, see Program Units and Procedures.

Internal procedures, like modules, provide a means of encapsulation. Where modules can be used to store routines commonly used by many programs, internal procedures separate functions and subroutines whose use is limited or temporary.

Storing Object Code in Static Libraries

Another way to organize source code used by several projects is to build a static library (for Windows* OS, `.lib` and for Linux* OS and Mac OS* X, `.a`) containing the object files for the reused procedures. You can create a static library by doing the following:

- From the Microsoft Visual Studio* integrated development environment (IDE), create and build a Fortran Static Library project type.
- From the command line, use the `ar` command (on Linux OS and Mac OS X) or the `LIB` command (on Windows OS).

After you have created a static library, you can use it as input to other types of Intel Fortran projects.

Storing Routines in Shareable Libraries

You can organize the code in your application by storing the executable code for certain routines in a shareable library (`.dll` for Windows* OS, `.so` for Linux* OS, `.dylib` for Mac OS* X). You can then build your applications so that they call these routines from the shareable library.

When routines in a shareable library are called, the routines are loaded into memory at run-time as they are needed. This is most useful when several applications use a common group of routines. By storing these common routines in a shareable library, you reduce the size of each application that calls the library. In addition, you can update the routines in the library without having to rebuild any of the applications that call the library.

Programming with Mixed Languages

13

Programming with Mixed Languages Overview

Mixed-language programming is the process of building programs in which the source code is written in two or more languages. It allows you to:

- Call existing code that is written in another language
- Use procedures that may be difficult to implement in a particular language
- Gain advantages in processing speeds

Mixed-language programming is possible between Intel® Fortran and other languages. Although other languages (such as assembly language) are discussed, the primary focus of this section is programming using Intel Fortran and C/C++ . Mixed language programming between these two languages is relatively straightforwardbecause each language implements functions, subroutines, and procedures in approximately the same way.

Calling Subprograms from the Main Program

Calls from the Main Program

The Intel® Fortran main program can call Intel Fortran subprograms, including subprograms in static and shared libraries.

For mixed-language applications, the Intel Fortran main program can call subprograms written in C/C++ if the appropriate calling conventions are used (see Calling C Procedures from a Fortran program).

Intel Fortran subprograms can be called by C/C++ main programs. If the main program is C/C++, you need to use the `-nofor_main` compiler option to indicate this.

Calls to the Subprogram

You can use subprograms in static libraries if the main program is written in Intel Fortran or C/C++.

You can use subprograms in shared libraries in mixed-language applications if the main program is written in Intel Fortran or C/C++.

Summary of Mixed-Language Issues

Mixed-language programming involves a call from a routine written in one language to a function, procedure, or subroutine written in another language. For example, a Fortran main program may need to execute a specific task that you want to program separately in an assembly-language procedure, or you may need to call an existing shared library or system procedure.

Mixed-language programming is possible with Intel® Fortran, Visual C/C++*, and Intel® C++, because each language implements functions, subroutines, and procedures in approximately the same way.

Intel Fortran includes several Fortran 2003 features that provide interoperability with C. An entity is considered interoperable if equivalent declarations can be made for it in both languages. Interoperability is provided for variables, derived types, and procedures.

The following features are supported:

- BIND attribute and statement, which specifies that an object is interoperable with C and has external linkage.
- language binding in FUNCTION and SUBROUTINE statements
- language binding in derived-type statements

For more information, see Interoperability with C.

Programming with Fortran and C/C++ Considerations

A summary of major Fortran and C/C++ mixed-language issues follows:

- Fortran and C implement functions and routines differently. For example, a C main program could call an external void function, which is actually implemented as a Fortran subroutine:

  Table 32: Language Equivalents for Calls to Routines

Language	Call with Return Value	Call with No Return Value
Fortran	FUNCTION	SUBROUTINE
C and C++	function	(void) function

- Generally, Fortran/C programs are mixed to allow one to use existing code written in the other language. Either Fortran or C can call the other, so the main routine can be in either language. On Linux OS and Mac OS X systems, if Fortran is not the main routine, the `-nofor-main` compiler option must be specified on the command line.

- To use the same Microsoft* visual development environment for multiple languages, you must have the same version of the visual development environment for your languages.
- Fortran adds an underscore to external names; C does not.
- Fortran changes the case of external names to lowercase; C leaves them in their original case.
- Fortran passes numeric data by reference; C passes by value.

NOTE. You can override some default Fortran behavior by using ATTRIBUTES and ALIAS. ATTRIBUTES C causes Fortran to act like C in external names and the passing of numeric data. ALIAS causes Fortran to use external names in their original case.

- Fortran subroutines are equivalent to C void routines.
- Fortran requires that the length of strings be passed; C is able to calculate the length based on the presence of a trailing null. Therefore, if Fortran is passing a string to a C routine, that string needs to be terminated by a null; for example:

```
"mystring"c or  StringVar // CHAR(0)
```

- For the following data types, Fortran adds a hidden first argument to contain function return values: COMPLEX, REAL*16, CHARACTER, and derived types.
- On Linux* systems, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.

For more information on mixed language programming using Intel Fortran and C, see the following:

- Compiling and Linking Intel Fortran/C Programs
- Calling C Procedures from an Intel Fortran Program

Programming with Fortran and Assembly-Language Considerations

A summary of Fortran/assembly language issues follows:

- Assembly-language routines can be small and can execute quickly because they do not require initialization as do high-level languages like Fortran and C.

- Assembly-language routines allow access to hardware instructions unavailable to the high-level language user. In a Fortran/assembly-language program, compiling the main routine in Fortran gives the assembly code access to Fortran high-level procedures and library functions, yet allows freedom to tune the assembly-language routines for maximum speed and efficiency. The main program can also be an assembly-language program.

Other Mixed-Language Programming Considerations

There are other important differences in languages; for instance, argument passing, naming conventions, and other interface issues must be thoughtfully and consistently reconciled between any two languages to prevent program failure and indeterminate results. However, the advantages of mixed-language programming often make the extra effort worthwhile. The remainder of this section provides an explanation of the techniques you can use to reconcile differences between Fortran and other languages.

Adjusting calling conventions, adjusting naming conventions and writing interface procedures are discussed in the following topics:

- Adjusting Calling Conventions in Mixed-Language Programming
- Adjusting Naming Conventions in Mixed-Language Programming
- Prototyping a Procedure in Fortran

After establishing a consistent interface between mixed-language procedures, you then need to reconcile any differences in the treatment of individual data types (strings, arrays, and so on). This is discussed in Exchanging and Accessing Data in Mixed-Language Programming. You also need to be concerned with data types, because each language handles them differently. This is discussed in Handling Data Types in Mixed-Language Programming.

NOTE. This section uses the term "routine" in a generic way, to refer to functions, subroutines, and procedures from different languages.

Adjusting Calling Conventions in Mixed-Language Programming

Adjusting Calling Conventions in Mixed-Language Programming Overview

The calling convention determines how a program makes a call to a routine, how the arguments are passed, and how the routines are named.

A calling convention includes:

- Stack considerations
  - Does a routine receive a varying or fixed number of arguments?
  - Which routine clears the stack after a call?
- Naming conventions
  - Is lowercase or uppercase significant or not significant?
  - Are external names altered?
- Argument passing protocol
  - Are arguments passed by value or by reference?
  - What are the equivalent data types and data structures among languages?

In a single-language program, calling conventions are nearly always correct, because there is one default for all routines and because header files or Fortran module files with interface blocks enforce consistency between the caller and the called routine.

In a mixed-language program, different languages cannot share the same header files. If you link Fortran and C routines that use different calling conventions, the error is not apparent until the bad call is made at run time. During execution, the bad call causes indeterminate results and/or a fatal error. The error, caused by memory or stack corruption due to calling errors, often occurs in a seemingly arbitrary place in the program.

The discussion of calling conventions between languages applies only to external procedures. You cannot call internal procedures from outside the program unit that contains them.

A calling convention affects programming in a number of ways:

1. The caller routine uses a calling convention to determine the order in which to pass arguments to another routine; the called routine uses a calling convention to determine the order in which to receive the arguments passed to it. In Fortran, you can specify these conventions in a mixed-language interface with the `INTERFACE` statement or in a data or function declaration. C/C++ and Fortran both pass arguments in order from left to right.
2. The caller routine and the called routine use a calling convention to select the option of passing a variable number of arguments.
3. The caller routine and the called routine use a calling convention to pass arguments by value (values passed) or by reference (addresses passed). Individual Fortran arguments can also be designated with `ATTRIBUTES` option VALUE or REFERENCE.

4. The caller routine and the called routine use a calling convention to establish naming conventions for procedure names. You can establish any procedure name you want, regardless of its Fortran name, with the ALIAS directive (or ATTRIBUTES option ALIAS). This is useful because C is case sensitive, while Fortran is not.

ATTRIBUTES Properties and Calling Conventions

The ATTRIBUTES properties (also known as options) C, STDCALL (Windows* OS only), REFERENCE, VALUE, and VARYING affect the calling convention of routines. You can specify:

- The C, STDCALL, REFERENCE, and VARYING properties for an entire routine.
- The VALUE and REFERENCE properties for individual arguments.

By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value). If the C (or, for Windows OS, STDCALL) option is used, the default changes to passing almost all data except arrays by value. However, in addition to the calling-convention options C and STDCALL, you can specify argument options, VALUE and REFERENCE, to pass arguments by value or by reference, regardless of the calling convention option. Arrays can only be passed by reference.

Different Fortran calling conventions can be specified by declaring the Fortran procedure to have certain attributes.

It is advisable to use the DECORATE option in combination with the ALIAS option to ensure appropriate name decoration regardless of operating system or architecture. The DECORATE option indicates that the external name specified in ALIAS should have the correct prefix and postfix decorations for the calling mechanism in effect.

Naming conventions are as follows:

- leading (prefix) underscore for Windows operating systems based on IA-32 architecture; no underscores for Windows operating systems based on Intel® 64 architecture and Windows systems based on IA-64 architecture.
- trailing (postfix) underscore for all Linux operating systems
- leading and trailing underscores for all Mac OS* X operating systems

For example:

```
   INTERFACE
      SUBROUTINE MY_SUB (I)
          !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'My_Sub'
 :: MY_SUB
          INTEGER I
      END SUBROUTINE MY_SUB
   END INTERFACE
```

This code declares a subroutine named `MY_SUB` with the C property. The external name will be appropriately decorated for the operating system and platform.

The following table summarizes the effect of the most common Fortran calling-convention directives.

Table 33: Calling Conventions for ATTRIBUTES Options

Argument	Default	C	C, REFERENCE	STDCALL (Windows OS IA-32 architecture)	STDCALL, REFERENCE (Windows OS IA-32 architecture)
Scalar	Reference	Value	Reference	Value	Reference
Scalar [value]	Value	Value	Value	Value	Value
Scalar [reference]	Reference	Reference	Reference	Reference	Reference
String	Reference, either Len:End or Len:Mixed	String(1:1)	Reference, either Len:End or Len:Mixed	String(1:1)	String(1:1)
String [value]	Error	String(1:1)	String(1:1)	String(1:1)	String(1:1)
String [reference]	Reference, either No Len or Len:Mixed	Reference, No Len	Reference, No Len	Reference, No Len	Reference, No Len
Array	Reference	Reference	Reference	Reference	Reference
Array [value]	Error	Error	Error	Error	Error
Array [reference]	Reference	Reference	Reference	Reference	Reference
Derived Type	Reference	Value, size dependent	Reference	Value, size dependent	Reference

Argument	Default	C	C, REFERENCE	STDCALL (Windows OS IA-32 architecture)	STDCALL, REFERENCE (Windows OS IA-32 architecture)
Derived Type [value]	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent	Value, size dependent
Derived Type [reference]	Reference	Reference	Reference	Reference	Reference
F90 Pointer	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
F90 Pointer [value]	Error	Error	Error	Error	Error
F90 Pointer [reference]	Descriptor	Descriptor	Descriptor	Descriptor	Descriptor
Naming Conventions					
Prefix	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_ (Windows operating systems using IA-32 architecture, Mac OS X operating systems) *none* for all others	_	_
Suffix	none (Windows OS) _ (Linux OS, Mac OS X)	*none*	*none*	@*n*	@*n*

Argument	Default	C	C, REFERENCE	STDCALL (Windows OS IA-32 architecture)	STDCALL, REFERENCE (Windows OS IA-32 architecture)
Case	Upper Case	Lower Case	Lower Case	Lower Case	Lower Case
Stack Cleanup	Caller	Caller	Caller	Callee	Callee

The terms in the above table mean the following:

[value] Argument assigned the VALUE attribute.

[reference] Argument assigned the REFERENCE attribute.

Value The argument value is pushed on the stack. All values are padded to the next 4-byte boundary.

Reference On systems using IA-32 architecture, the 4-byte argument address is pushed on the stack.

On systems using Intel® 64 and IA-64 architectures, the 8-byte argument address is pushed on the stack.

Len:End or Len:Mixed For certain string arguments:

- Len:End applies when -nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack after all of the other arguments. This is the default.
- Len:Mixed applies when -mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.

No Len or Len:Mixed For certain string arguments:

- No Len applies when nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg (Windows) is set. The length of the string is not available to the called procedure. This is the default.

	• Len:Mixed applies when mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg (Windows) is set. The length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
No Len	For string arguments, the length of the string is not available to the called procedure.
String(1:1)	For string arguments, the first character is converted to INTEGER(4) as in ICHAR(string(1:1)) and pushed on the stack by value.
Error	Produces a compiler error.
Descriptor	On systems using IA-32 architecture, the 4-byte address of the array descriptor. On systems using Intel® 64 architecture and systems using IA-64 architecture, the 8-byte address of the array descriptor.
@*n*	On systems using IA-32 architecture, the at sign (@) followed by the number of bytes (in decimal) required for the argument list.
Size dependent	On systems using IA-32 architecture, derived-type arguments specified by value are passed as follows: • Arguments from 1 to 4 bytes are passed by value. • Arguments from 5 to 8 bytes are passed by value in two registers (two arguments). • Arguments more than 8 bytes provide value semantics by passing a temporary storage address by reference.
Upper Case	Procedure name in all uppercase.
Lower Case	Procedure name in all lowercase.
Callee	The procedure being called is responsible for removing arguments from the stack before returning to the caller.
Caller	The procedure doing the call is responsible for removing arguments from the stack after the call is over.

The following table shows which Fortran ATTRIBUTES options match other language calling conventions.

Table 35: Matching Calling Conventions

Other Language Calling Convention	Matching ATTRIBUTES Option
C/C++ cdecl (default)	C
C/C++ __stdcall (Windows OS only)	STDCALL
MASM C (in PROTO and PROC declarations) (Windows OS only)	C
MASM STDCALL (in PROTO and PROC declarations) (Windows OS only)	STDCALL
Assembly (Linux OS only)	C

The ALIAS option can be used with any other Fortran calling-convention option to preserve mixed-case names. You can also use the DECORATE option in combination with the ALIAS option to specify that the external name specified in ALIAS should have the correct prefix and postfix decorations for the calling mechanism in effect.

For Windows operating systems, the compiler option `/iface` also establishes some default argument passing conventions. The `/iface` option has the following choices:

Option	How are arguments passed?	Append `@n` to names on systems using IA-32 architecture?	Who cleans up stack?	Varargs support?
/iface:cref	By reference	No	Caller	Yes
/iface:stdref	By reference	Yes	Callee	No
/iface:default	By reference	No	Caller	Yes
/iface:c	By value	No	Caller	Yes
/iface:stdcall	By value	Yes	Callee	No
/iface:cvf	By reference	Yes	Callee	No

Adjusting Naming Conventions in Mixed-Language Programming

Adjusting Naming Conventions in Mixed-Language Programming Overview

The ATTRIBUTES options C and, for Windows* OS, STDCALL, determine naming conventions as well as calling conventions.

Calling conventions specify how arguments are moved and stored; naming conventions specify how symbol names are altered when placed in an .OBJ file. Names are an issue for external data symbols shared among parts of the same program as well as among external routines. Symbol names (such as the name of a subroutine) identify a memory location that must be consistent among all calling routines.

Parameter names (names given in a procedure definition to variables that are passed to it) are never affected.

Names are altered because of case sensitivity (in C and MASM), lack of case sensitivity (in Fortran), name decoration (in C++), or other issues. If naming conventions are not reconciled, the program cannot successfully link and you will receive an "unresolved external" error.

C/C++ Naming Conventions

C and C++ preserve case sensitivity in their symbol tables while Fortran by default does not; this is a difference that requires attention. Fortunately, you can use the Fortran directive `ATTRIBUTES` ALIAS option to resolve discrepancies between names, to preserve mixed-case names, or to override the automatic case conversion of names by Fortran.

C++ uses the same calling convention and argument-passing techniques as C, but naming conventions differ because of C++ decoration of external symbols. When the C++ code resides in a `.cpp` file (created when you select C/C++ file from the integrated development environment), C++ name decoration semantics are applied to external names, often resulting in linker errors. The extern "C" syntax makes it possible for a C++ module to share data and routines with other languages by causing C++ to drop name decoration.

The following example declares `prn` as an external function using the C naming convention. This declaration appears in C++ source code:

```
extern "C" { void prn(); }
```

To call functions written in Fortran, declare the function as you would in C and use a "C" linkage specification. For example, to call the Fortran function FACT from C++, declare it as follows:

```
extern "C" { int fact( int* n ); }
```

The extern "C" syntax can be used to adjust a call from C++ to other languages, or to change the naming convention of C++ routines called from other languages. However, extern "C" can only be used from within C++. If the C++ code does not use extern "C" and cannot be changed,

you can call C++ routines only by determining the name decoration and generating it from the other language. Such an approach should only be used as a last resort, because the decoration scheme is not guaranteed to remain the same between versions.

Use of extern "C" has some restrictions:

- You cannot declare a member function with extern "C".
- You can specify extern "C" for only one instance of an overloaded function; all other instances of an overloaded function have C++ linkage.

Procedure Names for Fortran, C, C++, and MASM

The following table summarizes how Fortran, C/C++ and, for Windows, MASM handle procedure names. Note that for MASM, the table does not apply if the CASEMAP: ALL option is used.

Table 37: Naming Conventions in Fortran, C, Visual C++, and MASM

Language	Attributes	Name Translated As	Case of Name in .OBJ File
Fortran	*c***DEC$ ATTRIBUTES** C	*name* (Linux* OS) *_name* (Windows* OS) *_name* (Mac OS* X)	All lowercase
Fortran (Windows OS)	*c***DEC$ ATTRIBUTES** STDCALL	*_name*@*n*	All lowercase
Fortran	default	*name_* (Linux OS) *_name* (Windows OS) *_name_* (Mac OS X)	All uppercase
C	cdecl (default)	*name* (Linux OS) *_name* (Windows OS) *_name* (Mac OS X)	Mixed case preserved
C (Windows OS only)	__stdcall	*_name*@*n*	Mixed case preserved

Language	Attributes	Name Translated As	Case of Name in .OBJ File
C++	Default	*name*@@*decoration* (Linux OS) *_name*@@*decoration* (Windows OS) __decoration*name*decoration (Mac OS X)	Mixed case preserved
Linux OS, Mac OS X Assembly	Default	*name* (Linux OS) *_name* (Mac OS X)	Mixed case preserved
MASM (Windows OS)	C (in PROTO and PROC declarations)	*_name*	Mixed case preserved
MASM (Windows OS)	STDCALL (in PROTO and PROC declarations)	*_name*@*n*	Mixed case preserved

In the preceding table:

- The leading underscore (such as *_name*) is used on Windows operating systems based on IA-32 architecture only.
- @*n* represents the stack space, in decimal notation, occupied by parameters on Windows operating systems based on IA-32 architecture only.

For example, assume a function is declared in C as:

```
extern int __stdcall Sum_Up( int a, int b, int c );
```

Each integer occupies 4 bytes, so the symbol name placed in the .OBJ file on systems based on IA-32 architecture is:

```
_Sum_Up@12
```

On systems based on Intel® 64 architecture and those based on IA-64 architecture, the symbol name placed in the .OBJ file is:

```
Sum_Up
```

Reconciling the Case of Names

The following summarizes how to reconcile names between languages:

- All-Uppercase Names (default on Windows OS)

  If you call a Fortran routine that uses Fortran defaults and cannot recompile the Fortran code, then in C,you must use an all-uppercase name to make the call. In MASM you must either use an all-uppercase name or set the OPTION CASEMAP directive to ALL, which translates all identifiers to uppercase. Use of the __stdcall convention in C code or STDCALL in MASM PROTO and PROC declarations is not enough, because __stdcall and STDCALL always preserve case in these languages. Fortran generates all-uppercase names by default and the C or MASM code must match it.

  For example, these prototypes establish the Fortran function FFARCTAN(angle) where the argument angle has the `ATTRIBUTES` VALUE property:

  - In C:

    ```
    extern float FFARCTAN( float angle );
    ```

  - In MASM:

    ```
    .MODEL FLAT
    FFARCTAN PROTO, angle: REAL4
    ...
    FFARCTAN PROC, angle: REAL4
    ```

- All-Lowercase Names (default on Linux OS and Mac OS X)

  If the name of the routine appears as all lowercase in C or assembly, then naming conventions are automatically correct. Any case may be used in the Fortran source code, including mixed case since the name is changed to all lowercase.

  In Linux OS/Mac OS X Assembly, the following establishes the Fortran function `ffarctan`:

  ```
  #--Begin ffarctan_
  .globl ffarctan_
  ```

- Mixed-Case Names

  If the name of a routine appears as mixed-case in C or MASM and you need to preserve the case, use the Fortran `ATTRIBUTES` ALIAS option.

  To use the ALIAS option, place the name in quotation marks exactly as it is to appear in the object file.

  The following is an example for referring to the C function My_Proc:

  ```
  !DEC$ ATTRIBUTES DECORATE,ALIAS:'My_Proc'
   :: My_Proc
  ```

  This example uses DECORATE to automatically reconcile the external name for the target platform.

Fortran Module Names and ATTRIBUTES

Fortran module entities (data and procedures) have external names that differ from other external entities. Module names use the convention:

```
modulename_mp_entity_(Linux OS and Mac OS X)
_MODULENAME_mp_ENTITY [ @stacksize ] (Windows OS)
```

modulename is the name of the module. For Windows* operating systems, the name is uppercase by default.

`entity` is the name of the module procedure or module data contained within *MODULENAME*. For Windows* operating systems, *ENTITY* is uppercase by default.

*_mp_* is the separator between the module and entity names and is always lowercase.

For example:

```
    MODULE mymod
      INTEGER a
   CONTAINS
      SUBROUTINE b (j)
         INTEGER j
      END SUBROUTINE
   END MODULE
```

This results in the following symbols being defined in the compiled object file on Linux operating systems (On Mac OS X operating systems, the symbols would begin with an underscore)::

```
mymod_mp_a_
mymod_mp_b_
```

The following symbols are defined in the compiled object file on Windows operating systems based on IA-32 architecture:

```
_MYMOD_mp_A
_MYMOD_mp_B
```

Compiler options can affect the naming of module data and procedures.

NOTE. Except for ALIAS, `ATTRIBUTES` options do not affect the module name.

The following table shows how each ATTRIBUTES option affects the subroutine in the previous example module.

Table 38: Effect of ATTRIBUTES options on Fortran Module Names

ATTRIBUTES Option Given to Routine 'b'	Procedure Name in .OBJ file on Systems Using IA-32 Architecture	Procedure Name in .OBJ file on Systems Using Intel® 64 Architecture and IA-64 Architecture
None	`mymod_mp_b_` (Linux OS) `_mymod_mp_b_` (Mac OS X) `_MYMOD_mp_B` (Windows OS)	`mymod_mp_b_` (Linux OS) `_mymod_mp_b_` (Mac OS X) `MYMOD_mp_B` (Windows OS)
C	`mymod_mp_b_` (Linux OS) `_mymod_mp_b_` (Mac OS X) `MYMOD_mp_b` (Windows OS)	`mymod_mp_b` (Linux OS) `_mymod_mp_b` (Mac OS X) `MYMOD_mp_b` (Windows OS)
STDCALL (Windows OS only)	`_MYMOD_mp_b@4`	`MYMOD_mp_b`
ALIAS	Overrides all others, name as given in the alias	Overrides all others, name as given in the alias
VARYING	No effect on name	No effect on name

You can write code to call Fortran modules or access module data from other languages. As with other naming and calling conventions, the module name must match between the two languages. Generally, this means using the C or STDCALL convention in Fortran, and if defining a module in another language, using the ALIAS option to match the name within Fortran. For examples, see Using Modules in Mixed-Language Programming.

Prototyping a Procedure in Fortran

You define a prototype (interface block) in your Fortran source code to tell the Fortran compiler which language conventions you want to use for an external reference. The interface block is introduced by the `INTERFACE` statement. See Program Units and Procedures for a more detailed description of the `INTERFACE` statement.

The general form for the INTERFACE statement is:

```
INTERFACE
routine statement
[routine ATTRIBUTE options]
[argument ATTRIBUTE options]
formal argument declarations
END routine name
END INTERFACE
```

The *routine statement* defines either a FUNCTION or a SUBROUTINE, where the choice depends on whether a value is returned or not, respectively. The optional *routine* ATTRIBUTE options (such as C, and, for Windows, STDCALL) determine the calling, naming, and argument-passing conventions for the routine in the prototype statement. The optional *argument* ATTRIBUTE options (such as VALUE and REFERENCE) are properties attached to individual arguments. The *formal argument declarations* are Fortran data type declarations. Note that the same INTERFACE block can specify more than one procedure.

For example, suppose you are calling a C function that has the following prototype:

```
extern void My_Proc (int i);
```

The Fortran call to this function should be declared with the following INTERFACE block:

```
INTERFACE
  SUBROUTINE my_Proc (I)
   !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'My_Proc' :: my_Proc
   INTEGER I
  END SUBROUTINE my_Proc
END INTERFACE
```

Note that, except in the ALIAS string, the case of My_Proc in the Fortran program does not matter.

Exchanging and Accessing Data in Mixed-Language Programming

Exchanging and Accessing Data in Mixed-Language Programming

You can use several approaches to sharing data between mixed-language routines, which can be used within the individual languages as well.

Generally, if you have a large number of parameters to work with or you have a large variety of parameter types, you should consider using modules or external data declarations. This is true when using any given language, and to an even greater extent when using mixed languages.

See Also

- Exchanging and Accessing Data in Mixed-Language Programming

- Passing Arguments in Mixed-Language Programming
- Using Modules in Mixed-Language Programming
- Using Common External Data in Mixed-Language Programming

Passing Arguments in Mixed-Language Programming

You can pass data between Fortran and C, C++, and MASM through calling argument lists just as you can within each language (for example, the argument list a, b and c in CALL MYSUB(a,b,c)). There are two ways to pass individual arguments:

- *By value*, which passes the argument's value.
- *By reference*, which passes the address of the arguments. On systems based on IA-32 architecture, Fortran, C, and C++ use 4-byte addresses. On systems based on Intel® 64 architecture and those based on IA-64 architecture, these languages use 8-byte addresses.

You need to make sure that for every call, the calling program and the called routine agree on how each argument is passed. Otherwise, the called routine receives bad data.

The Fortran technique for passing arguments changes depending on the calling convention specified. By default, Fortran passes all data by reference (except the hidden length argument of strings, which is passed by value).

If the ATTRIBUTES C option or, for Windows OS, the STDCALL option is used, the default changes to passing all data by value except arrays. If the procedure has the REFERENCE option as well as the C or STDCALL option, all arguments by default are passed by reference.

You can specify also argument options, VALUE and REFERENCE, to pass arguments by value or by reference. In mixed-language programming, it is a good idea to specify the passing technique explicitly rather than relying on defaults.

NOTE. On Windows operating systems, the compiler option `/iface` also establishes some default argument passing conventions (such as for hidden length of strings). See ATTRIBUTES Properties and Calling Conventions.

Examples of passing by reference and value for C and MASM follow. All are interfaces to the example Fortran subroutine `TESTPROC` below. The definition of `TESTPROC` declares how each argument is passed. The REFERENCE option is not strictly necessary in this example, but using it makes the argument's passing convention conspicuous.

```
SUBROUTINE TESTPROC( VALPARM, REFPARM )
  !DEC$ ATTRIBUTES VALUE :: VALPARM
  !DEC$ ATTRIBUTES REFERENCE :: REFPARM
  INTEGER VALPARM
  INTEGER REFPARM
END SUBROUTINE
```

In C and C++, all arguments are passed by value, except arrays, which are passed by reference to the address of the first member of the array. Unlike Fortran, C and C++ do not have calling-convention directives to affect the way individual arguments are passed. To pass non-array C data by reference, you must pass a pointer to it. To pass a C array by value, you must declare it as a member of a structure and pass the structure. The following C declaration sets up a call to the example Fortran `TESTPROC` subroutine:

For Linux OS:

```
extern void testproc_(
 int ValParm, int *RefParm );
```

For Windows OS:

```
extern void TESTPROC(
 int ValParm, int *RefParm );
```

In MASM (Windows OS only), arguments are passed by value by default. Arguments to be passed by reference are designated with PTR in the PROTO and PROC directives. For example:

```
TESTPROC PROTO, valparm:
 SDWORD, refparm: PTR SDWORD
```

To use an argument passed by value, use the value of the variable. For example:

```
mov eax, valparm ; Load value of argument
```

This statement places the value of valparm into the EAX register.

To use an argument passed by reference, use the address of the variable. For example:

```
mov ecx, refparm ; Load address of argument
mov eax, [ecx]   ; Load value of argument
```

These statements place the value of refparm into the EAX register.

The following table summarizes how to pass arguments by reference and value. An array name in C is equated to its starting address because arrays are normally passed by reference. You can assign the REFERENCE property to a procedure, as well as to individual arguments.

Table 39: Passing Arguments by Reference and Value

Language	ATTRIBUTE	Argument Type	To Pass by Reference	To Pass by Value
Fortran	Default	Scalars and derived types	Default	VALUE option
	C (or, for Windows OS, STDCALL) option	Scalars and derived types	REFERENCE option	Default
	Default	Arrays	Default	Cannot pass by value

Language	ATTRIBUTE	Argument Type	To Pass by Reference	To Pass by Value
	C (or, for Windows OS, STDCALL) option	Arrays	Default	Cannot pass by value
C/C++		Non-arrays	Pointer argument_name	Default
		Arrays	Default	Struct {type} array_name
Assembler MASM (Windows OS only)		All types	PTR	Default

This table does not describe argument passing of strings and Fortran 95/90 pointer arguments in Intel Fortran, which are constructed differently than other arguments. By default, Fortran passes strings by reference along with the string length. String length placement depends on whether the compiler option `-mixed-str-len-arg` (Linux OS and Mac OS X) or `/iface:mixed_str_len_arg` (Windows OS) is set immediately after the address of the beginning of the string. It also depends on whether -nomixed-str-len-arg (Linux OS and Mac OS X) or `/iface:nomixed_str_len_arg` (Windows OS) is set after all arguments.

Fortran 95/90 array pointers and assumed-shape arrays are passed by passing the address of the array descriptor.

For a discussion of the effect of attributes on passing Fortran 95/90 pointers and strings, see Handling Fortran Array Pointers and Allocatable Arrays and Handling Character Strings.

See Also

- Exchanging and Accessing Data in Mixed-Language Programming
- Handling Character Strings
- Handling Numeric, Complex, and Logical Data Types

Using Modules in Mixed-Language Programming

Modules are the simplest way to exchange large groups of variables with C, because Intel Fortran modules are directly accessible from C/C++. The following example declares a module in Fortran, then accesses its data from C.

The Fortran code:

```
! F90 Module definition
   MODULE EXAMP
      REAL A(3)
```

```
      INTEGER I1, I2
      CHARACTER(80) LINE
      TYPE MYDATA
         SEQUENCE
         INTEGER N
         CHARACTER(30) INFO
      END TYPE MYDATA
   END MODULE EXAMP
```

The C code:

```
\* C code accessing module data *\
extern float EXAMP_mp_A[3];
extern int EXAMP_mp_I1, EXAMP_mp_I2;
extern char EXAMP_mp_LINE[80];
extern struct {
          int N;
          char INFO[30];
} EXAMP_mp_MYDATA;
```

When the C++ code resides in a `.cpp` file, C++ semantics are applied to external names, often resulting in linker errors. In this case, use the extern "C" syntax (see C/C++ Naming Conventions):

```
\* C code accessing module data in .cpp file*\
extern "C" float EXAMP_mp_A[3];
extern "C" int EXAMP_mp_I1, EXAMP_mp_I2;
extern "C" char EXAMP_mp_LINE[80];
extern "C" struct {
          int N;
          char INFO[30];
} EXAMP_mp_MYDATA;
```

You can define an interface to a C routine in a module, then use it like you would an interface to a Fortran routine. The C code is:

The C code:

```
 // C procedure
void pythagoras (float a, float b, float *c)
{
   *c = (float) sqrt(a*a + b*b);
}
```

When the C++ code resides in a `.cpp` file, use the extern "C" syntax (see C/C++ Naming Conventions):

```
// C procedure
extern "C" void pythagoras (float a, float b, float *c)
{
   *c = (float) sqrt(a*a + b*b);
}
```

The following Fortran code defines the module CPROC:

```
 ! Fortran 95/90 Module including procedure
   MODULE CPROC
      INTERFACE
         SUBROUTINE PYTHAGORAS (a, b, res)
          !DEC$ ATTRIBUTES C :: PYTHAGORAS
          !DEC$ ATTRIBUTES REFERENCE :: res
! res is passed by REFERENCE because its individual attribute
! overrides the subroutine's C attribute
            REAL a, b, res
! a and b have the VALUE attribute by default because
! the subroutine has the C attribute
         END SUBROUTINE
      END INTERFACE
   END MODULE
```

The following Fortran code calls this routine using the module CPROC:

```
 ! Fortran 95/90 Module including procedure
   USE CPROC
      CALL PYTHAGORAS (3.0, 4.0, X)
      TYPE *,X
   END
```

Using Common External Data in Mixed-Language Programming

Common external data structures include Fortran common blocks, and C structures and variables that have been declared global or external. All of these data specifications create external variables, which are variables available to routines outside the routine that defines them.

This section applies only to Fortran/C and, on Windows, Fortran/MASM mixed-language programs.

External variables are case sensitive, so the cases must be matched between different languages, as discussed in the section on naming conventions. Common external data exchange is described in the following sections:

- Using Global Variables
- Using Fortran Common Blocks and C Structures

Using Global Variables in Mixed-Language Programming

A variable can be shared between Fortran and C or MASM by declaring it as global (or `COMMON`) in one language and accessing it as an external variable in the other language.

In Fortran, a variable can access a global parameter by using the EXTERN option for ATTRIBUTES. For example:

```
    !DEC$ ATTRIBUTES C, EXTERN :: idata
   INTEGER idata (20)
```

EXTERN tells the compiler that the variable is actually defined and declared global in another source file. If Fortran declares a variable external with EXTERN, the language it shares the variable with must declare the variable global.

In C, a variable is declared global with the statement:

```
  int idata[20]; // declared as global (outside of any function)
```

MASM declares a parameter global (PUBLIC) with the syntax:

```
            PUBLIC [langtype] name
```

where *name* is the name of the global variable to be referenced, and the optional *langtype* is STDCALL or C. The option *langtype*, if present, overrides the calling convention specified in the .MODEL directive.

Conversely, Fortran can declare the variable global (COMMON) and other languages can reference it as external:

```
 ! Fortran declaring PI global
  REAL PI
  COMMON /PI/ PI ! Common Block and variable have the same name
```

In C, the variable is referenced as an external with the statement:

```
  //C code with external reference to PI
  extern float PI;
```

Note that the global name C references is the name of the Fortran common block, not the name of a variable within a common block. Thus, you cannot use blank common to make data accessible between C and Fortran. In the preceding example, the common block and the variable have the same name, which helps keep track of the variable between the two languages. Obviously, if a common block contains more than one variable they cannot all have the common block name. (See Using Fortran Common Blocks and C Structures.)

MASM and Assembly can also access Fortran global (COMMON) parameters with the ATTRIBUTES EXTERN directive. The syntax is:

```
EXTERN [langtype] name
```

where *name* is the name of the global variable to be referenced, and the optional *langtype* is STDCALL or C.

Using Fortran Common Blocks and C Structures

To reference C structures from Fortran common blocks and vice versa, you must take into account how common blocks and structures differ in their methods of storing member variables in memory. Fortran places common block variables into memory in order as close together as possible, with the following rules:

- A single BYTE, INTEGER(1), LOGICAL(1), or CHARACTER variable in common block list begins immediately following the previous variable or array in memory.
- All other types of single variables begin at the next even address immediately following the previous variable or array in memory.
- All arrays of variables begin on the next even address immediately following the previous variable or array in memory, except for CHARACTER arrays which always follow immediately after the previous variable or array.
- All common blocks begin on a four-byte aligned address.

Because of these padding rules, you must consider the alignment of C structure elements with Fortran common block elements and assure matching either by making all variables exactly equivalent types and kinds in both languages (using only 4-byte and 8-byte data types in both languages simplifies this) or by using the C pack pragmas in the C code around the C structure to make C data packing like Fortran data packing. For example:

```
#pragma pack(2)
struct {
       int N;
       char INFO[30];
} examp;
#pragma pack()
```

To restore the original packing, you must add #pragma pack( ) at the end of the structure. (Remember: Fortran module data can be shared directly with C structures with appropriate naming.)

Once you have dealt with alignment and padding, you can give C access to an entire common block or set of common blocks. Alternatively, you can pass individual members of a Fortran common block in an argument list, just as you can any other data item. Use of common blocks for mixed-language data exchange is discussed in the following sections:

- Accessing Common Blocks and C Structures Directly
- Passing the Address of a Common Block

Accessing Common Blocks and C Structures Directly

You can access Fortran common blocks directly from C by defining an external C structure with the appropriate fields, and making sure that alignment and padding between Fortran and C are compatible. The C and ALIAS ATTRIBUTES options can be used with a common block to allow mixed-case names.

As an example, suppose your Fortran code has a common block named Really, as shown:

```
  !DEC$ ATTRIBUTES ALIAS:'Really' :: Really
 REAL(4) x, y, z(6)
 REAL(8) ydbl
 COMMON / Really / x, y, z(6), ydbl
```

You can access this data structure from your C code with the following external data structures:

```
#pragma pack(2)
extern struct {
 float x, y, z[6];
 double ydbl;
} Really;
#pragma pack()
```

You can also access C structures from Fortran by creating common blocks that correspond to those structures. This is the reverse case from that just described. However, the implementation is the same because after common blocks and structures have been defined and given a common address (name), and assuming the alignment in memory has been dealt with, both languages share the same memory locations for the variables.

Passing the Address of a Common Block

To pass the address of a common block, simply pass the address of the first variable in the block, that is, pass the first variable by reference. The receiving C or C++ module should expect to receive a structure by reference.

In the following example, the C function initcb receives the address of a common block with the first variable named n, which it considers to be a pointer to a structure with three fields:

Fortran source code:

```
!
  INTERFACE
    SUBROUTINE initcb (BLOCK)
      !DEC$ ATTRIBUTES C :: initcb
      !DEC$ ATTRIBUTES REFERENCE :: BLOCK
      INTEGER BLOCK
    END SUBROUTINE
  END INTERFACE
```

```
!
  INTEGER n
  REAL(8) x, y
  COMMON /CBLOCK/n, x, y
   . . .
  CALL initcb( n )
```

C source code:

```
//
#pragma pack(2)
struct block_type
{
 int n;
 double x;
 double y;
};
#pragma pack()
//
void initcb( struct block_type *block_hed )
 {
  block_hed->n = 1;
  block_hed->x = 10.0;
  block_hed->y = 20.0;
}
```

Handling Data Types in Mixed-Language Programming

Handling Data Types in Mixed-Language Programming Overview

Even when you have reconciled calling conventions, naming conventions, and methods of data exchange, you must still be concerned with data types, because each language handles them differently. The following table lists the equivalent data types among Fortran, C, and MASM:

Table 40: Equivalent Data Types

Fortran Data Type	C Data Type	MASM Data Type
REAL(4)	float	REAL4
REAL(8)	double	REAL8
REAL(16)	---	---
CHARACTER(1)	unsigned char	BYTE
CHARACTER*(*)	See Handling Character Strings	

Fortran Data Type	C Data Type	MASM Data Type

COMPLEX(4)	struct complex4 {
	float real, imag;
	};
		COMPLEX4 STRUCT 4
		real REAL4 0
		imag REAL4 0
		COMPLEX4 ENDS

COMPLEX(8)	struct complex8 {
	double real, imag;
	};
		COMPLEX8 STRUCT 8
		real REAL8 0
		imag REAL8 0
		COMPLEX8 ENDS

COMPLEX(16)	---	---
All LOGICAL types	Use integer types for C, MASM	
INTEGER(1)	char	.sbyte
INTEGER(2)	short	.sword
INTEGER(4)	int	.sdword
INTEGER(8)	_int64	.qword

The following sections describe how to reconcile data types between the different languages:

- Handling Numeric, Complex, and Logical Data Types
- Handling Fortran 90 Array Pointers and Allocatable Arrays
- Handling Integer Pointers
- Handling Arrays and Fortran Array Descriptors
- Handling Character Strings
- Handling User-Defined Types

Handling Numeric, Complex, and Logical Data Types

Normally, passing numeric data does not present a problem. If a C program passes an unsigned data type to a Fortran routine, the routine can accept the argument as the equivalent signed data type, but you should be careful that the range of the signed type is not exceeded.

The table of Equivalent Data Types summarizes equivalent numeric data types for Fortran, MASM, and C/C++.

C, C++, and MASM do not directly implement the Fortran types COMPLEX(4), COMPLEX(8), and COMPLEX(16). However, you can write structures that are equivalent. The type COMPLEX(4) has two fields, both of which are 4-byte floating-point numbers; the first contains the real-number component, and the second contains the imaginary-number component. The type COMPLEX is equivalent to the type COMPLEX(4). The types COMPLEX(8) and COMPLEX(16) are similar except that each field contains an 8-byte or 16-byte floating-point number respectively.

NOTE. On systems based on IA-32 architecture, Fortran functions of type COMPLEX place a hidden COMPLEX argument at the beginning of the argument list. C functions that implement such a call from Fortran must declare this hidden argument explicitly, and use it to return a value. The C return type should be void.

Following are the C/C++ structure definitions for the Fortran COMPLEX types:

```
struct complex4 {
  float real, imag;
};
struct complex8 {
  double real, imag;
};
```

The MASM structure definitions for the Fortran COMPLEX types follow:

```
 COMPLEX4 STRUCT 4
  real REAL4 0
  imag REAL4 0
 COMPLEX4 ENDS
 COMPLEX8 STRUCT 8
  real REAL8 0
  imag REAL8 0
 COMPLEX8 ENDS
```

A Fortran LOGICAL(2) is stored as a 2-byte indicator value (0=false, and the `-fpscomp [no]logicals` (Linux* OS and Mac OS* X)or `/fpscomp:[no]logicals` (Windows* OS) compiler option determines how true values are handled). A Fortran LOGICAL(4) is stored as a 4-byte indicator value, and LOGICAL(1) is stored as a single byte. The type LOGICAL is the same as LOGICAL(4), which is equivalent to type int in C.

You can use a variable of type LOGICAL in an argument list, module, common block, or global variable in Fortran and type int in C for the same argument. Type LOGICAL(4) is recommended instead of the shorter variants for use in common blocks.

The C++ class type has the same layout as the corresponding C struct type, unless the class defines virtual functions or has base classes. Classes that lack those features can be passed in the same way as C structures.

Returning Complex Type Data

If a Fortran program expects a procedure to return a `COMPLEX` or `DOUBLE COMPLEX` value, the Fortran compiler adds an additional argument to the beginning of the called procedure argument list. This additional argument is a pointer to the location where the called procedure must store its result.

The example below shows the Fortran code for returning a complex data type procedure called `WBAT` and the corresponding C routine.

Example of Returning Complex Data Types from C to Fortran

Fortran code

```
COMPLEX BAT, WBAT
REAL X, Y
BAT = WBAT ( X, Y )
```

Corresponding C Routine

Linux example:

```
struct _mycomplex { float real;
 float imag; };
typedef struct _mycomplex _single_complex;

void wbat_ (_single_complex *location, float *x, float *y){
*location->real = *x;
*location->imag = *y;
return;
}
```

Windows OS example:

```
struct _mycomplex { float real, imag };
typedef struct _mycomplex _single_complex;

void WBAT (_single_complex location, float *x, float *y)
{
float realpart;
float imaginarypart;
... program text, producing realpart and
imaginarypart...
*location.real = realpart;
*location.imag = imaginarypart;
}
```

In the above example, the following restrictions and behaviors apply:

- The argument `location` does not appear in the Fortran call; it is added by the compiler.

- The C subroutine must copy the result's real and imaginary parts correctly into location.
- The called procedure is type void.

If the function returned a `DOUBLE COMPLEX` value, the type `float` would be replaced by the type `double` in the definition of `location` in `WBAT`.

Handling Fortran Array Pointers and Allocatable Arrays

The following affects how Fortran 95/90 array pointers and arrays are passed:

- the `ATTRIBUTES` properties in effect
- the `INTERFACE`, if any, of the procedure they are passed to>

If the `INTERFACE` declares the array pointer or array with deferred shape (for example, `ARRAY(:)`), its descriptor is passed. This is true for array pointers and all arrays, not just allocatable arrays. If the `INTERFACE` declares the array pointer or array with fixed shape, or if there is no interface, the array pointer or array is passed by base address as a contiguous array, which is like passing the first element of an array for contiguous array slices.

When a Fortran 95/90 array pointer or array is passed to another language, either its descriptor or its base address can be passed.

The following shows how allocatable arrays and Fortran 95/90 array pointers are passed with different attributes in effect:

- If the property of the array pointer or array is not included or is REFERENCE, it is passed by descriptor, regardless of the property of the passing procedure .
- If the property of the array pointer or array is VALUE, an error is returned, regardless of the property of the passing procedure.

Note that the VALUE option cannot be used with descriptor-based arrays.

When you pass a Fortran array pointer or an array by descriptor to a non-Fortran routine, that routine needs to know how to interpret the descriptor. Part of the descriptor is a pointer to address space, as a C pointer, and part of it is a description of the pointer or array properties, such as its rank, stride, and bounds.

For information about the Intel Fortran array descriptor format, see Handling Arrays and Fortran Array Descriptors.

Fortran 95/90 pointers that point to scalar data contain the address of the data and are not passed by descriptor.

Handling Integer Pointers

Intel® Fortran integer pointers (also known as Cray*-style pointers) are not the same as Fortran 90 pointers, but are instead like C pointers. On systems based on IA-32 architecture, integer pointers are 4-byte `INTEGER` quantities. On systems based on Intel® 64 architecture and those based on IA-64 architecture, integer pointers are 8-byte `INTEGER` quantities.

When passing an integer pointer to a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type.
- The argument passed from the Fortran routine should be the integer pointer name, not the pointee name.

Fortran main program:

```
! Fortran main program.
INTERFACE
SUBROUTINE Ptr_Sub (p)
!DEC$ ATTRIBUTES C, DECORATE, ALIAS:'Ptr_Sub' :: Ptr_Sub
INTEGER (KIND=INT_PTR_KIND()) p
END SUBROUTINE Ptr_Sub
END INTERFACE
REAL A(10), VAR(10)
POINTER (p, VAR) ! VAR is the pointee
! p is the integer pointer
p = LOC(A)
CALL Ptr_Sub (p)
WRITE(*,*) 'A(4) = ', A(4)
END
!
```

On systems using Intel® 64 architecture and IA-64 architecture, the declaration for p in the INTERFACE block is equivalent to INTEGER(8) p and on systems using IA-32 architecture, it is equivalent to INTEGER (4) p.

C subprogram:

```
   //C subprogram
void Ptr_Sub (float *p)
{
p[3] = 23.5;
}
```

When the main Fortran program and C function are built and executed, the following output appears:

```
A(4) = 23.50000
```

When receiving a pointer from a routine written in another language:

- The argument should be declared in the non-Fortran routine as a pointer of the appropriate data type and passed as usual.
- The argument received by the Fortran routine should be declared as an integer pointer name and the `POINTER` statement should associate it with a pointee variable of the appropriate data type (matching the data type of the passing routine). When inside the Fortran routine, use the pointee variable to set and access what the pointer points to.

Fortran subroutine:

```
! Fortran subroutine.
    SUBROUTINE Iptr_Sub (p)
    !DEC$ ATTRIBUTES C, DECORATE, ALIAS:'Iptr_Sub' :: Iptr_Sub
     INTEGER (KIND=INT_PTR_KIND()) p
       integer VAR(10)
       POINTER (p, VAR)
       OPEN (8, FILE='STAT.DAT')
       READ (8, *) VAR(4) ! Read from file and store the
                          ! fourth element of VAR
    END SUBROUTINE Iptr_Sub
!
//C main program
extern void Iptr_Sub(int *p);
main ( void )
```

C Main Program:

```
//C main program
extern void Iptr_Sub(int *p);
main ( void )
{
int a[10];
Iptr_Sub (&a[0]);
printf("a[3] = %i\n", a[3]);
}
```

When the main C program and Fortran subroutine are built and executed, the following output appears if the `STAT.DAT` file contains 4:

```
a[3] = 4
```

Handling Arrays and Fortran Array Descriptors

Fortran 95/90 allows arrays to be passed as array elements, as array subsections, or as whole arrays referenced by array name. Within Fortran, array elements are ordered in column-major order, meaning the subscripts of the lowest dimensions vary first.

When using arrays between Fortran and another language, differences in element indexing and ordering must be taken into account. You must reference the array elements individually and keep track of them. Array indexing is a source-level consideration and involves no difference in the underlying data.

Fortran and C arrays differ in two ways:

- The value of the lower array bound is different. By default, Fortran indexes the first element of an array as 1. C and C++ index it as 0. Fortran subscripts should therefore be one higher. (Fortran also provides the option of specifying another integer lower bound.)
- In arrays of more than one dimension, Fortran varies the left-most index the fastest, while C varies the right-most index the fastest. These are sometimes called column-major order and row-major order, respectively.

In C, the first four elements of an array declared as X[3][3] are:

```
X[0][0] X[0][1] X[0][2] X[1][0]
```

In Fortran, the first four elements are:

```
X(1,1) X(2,1) X(3,1) X(1,2)
```

The order of indexing extends to any number of dimensions you declare. For example, the C declaration:

```
int arr1[2][10][15][20];
```

is equivalent to the Fortran declaration:

```
INTEGER arr1( 20, 15, 10, 2 )
```

The constants used in a C array declaration represent extents, not upper bounds as they do in other languages. Therefore, the last element in the C array declared as int arr[5][5] is arr[4][4], not arr[5][5].

The following table shows equivalencies for array declarations.

Language	Array Declaration	Array Reference from Fortran
Fortran	`DIMENSION` x(i, k) -or- *type* x(i, k)	x(i, k)
C/C++	*type* x[k] [i]	x(i -1, k -1)

Handling Arrays in Visual Basic and MASM

The following information on Visual Basic and MASM applies to Windows operating systems only.

To pass an array from Visual Basic to Fortran, pass the first element of the array. By default, Visual Basic passes variables by reference, so passing the first element of the array will give Fortran the starting location of the array, just as Fortran expects. Visual Basic indexes the first array element as 0 by default, while Fortran by default indexes it as 1. Visual Basic indexing can be set to start with 1 using the statement:

```
Option Base 1
```

Alternatively, in the array declaration in either language you can set the array lower bound to any integer in the range -32,768 to 32,767. For example:

```
' In Basic
Declare Sub FORTARRAY Lib "fortarr.dll" (Barray as Single)
DIM barray (1 to 3, 1 to 7) As Single
Call FORTARRAY(barray (1,1))
! In Fortran
Subroutine FORTARRAY(arr)
  REAL arr(1:3,1:7)
```

In MASM, arrays are one-dimensional and array elements must be referenced byte-by-byte. The assembler stores elements of the array consecutively in memory, with the first address referenced by the array name. You then access each element relative to the first, skipping the total number of bytes of the previous elements. For example:

```
xarray    REAL4     1.1, 2.2, 3.3, 4.4 ; initializes
                     ; a four element array with
                     ; each element 4 bytes
```

Referencing xarray in MASM refers to the first element, the element containing 1.1. To refer to the second element, you must refer to the element 4 bytes beyond the first with xarray[4] or xarray+4. Similarly:

```
yarray     BYTE      256 DUP     ; establishes a
               ; 256 byte buffer, no initialization
zarray SWORD 100 DUP(0) ; establishes 100
               ; two-byte elements, initialized to 0
```

Intel Fortran Array Descriptor Format

For cases where Fortran 95/90 needs to keep track of more than a pointer memory address, the Intel Fortran Compiler uses an *array descriptor*, which stores the details of how an array is organized.

When using an explicit interface (by association or procedure interface block), Intel Fortran generates a descriptor for the following types of array arguments:

- Pointers to arrays (array pointers)
- Assumed-shape arrays
- Allocatable array

Certain data structure arguments do not use a descriptor, even when an appropriate explicit interface is provided. For example, explicit-shape and assumed-size arrays do not use a descriptor. In contrast, array pointers and allocatable arrays use descriptors regardless of whether they are used as arguments.

When calling between Intel Fortran and a non-Fortran language (such as C), using an *implicit* interface allows the array argument to be passed *without* an Intel Fortran descriptor. However, for cases where the called routine needs the information in the Intel Fortran descriptor, declare the routine with an *explicit* interface and specify the dummy array as either an assumed-shape array or with the pointer attribute.

You can associate a Fortran 95/90 pointer with any piece of memory, organized in any way desired (so long as it is "rectangular" in terms of array bounds). You can also pass Fortran 95/90 pointers to other languages, such as C, and have the other language correctly interpret the descriptor to obtain the information it needs.

However, using array descriptors can increase the opportunity for errors and the corresponding code is not portable. In particular, be aware of the following:

- If the descriptor is not defined correctly, the program may access the wrong memory address, possibly causing a General Protection Fault.
- Array descriptor formats are specific to each Fortran compiler. Code that uses array descriptors is *not*portable to other compilers or platforms. For example, the current Intel Fortran array descriptor format differs from the array descriptor format for Intel Fortran 7.0.
- The array descriptor format may change in the future.
- If the descriptor was built by the compiler, it cannot be modified by the user. Changing fields of existing descriptors is illegal.

The components of the current Intel Fortran array descriptor on systems using IA-32 architecture are as follows:

- The first longword (bytes 0 to 3) contains the base address. The base address plus the offset defines the first memory location (start) of the array.

- The second longword (bytes 4 to 7) contains the size of a single element of the array.
- The third longword (bytes 8 to 11) contains the A0 offset. The A0 offset is added to the base address to calculate the address for the element with all indices zero, even if that is outside the bounds of the actual array. This is helpful in computing array element addresses.
- The fourth longword (bytes 12 to 15) contains a set of flags used to store information about the array. This includes:
  - bit 1 (0x01): array is defined -- set if the array has been defined (storage allocated)
  - bit 2 (0x02): no deallocation allowed -- set if the array pointed to cannot be deallocated (that is, it is an explicit array target)
  - bit 3 (0x04): array is contiguous -- set if the array pointed to is a contiguous whole array or slice.
- The fifth longword (bytes 16 to 19) contains the number of dimensions (rank) of the array.
- The sixth longword (bytes 20 to 23) is reserved and should not be explicitly set.
- The remaining longwords (bytes 24 to 107) contain information about each dimension (up to seven). Each dimension is described by three additional longwords:
  - The number of elements (extent)
  - The distance between the starting address of two successive elements in this dimension, in bytes.
  - The lower bound

An array of rank one requires three additional longwords for a total of nine longwords (6 + 3*1) and ends at byte 35. An array of rank seven is described in a total of 27 longwords (6 + 3*7) and ends at byte 107.

For example, consider the following declaration:

```
 integer,target :: a(10,10)
integer,pointer :: p(:,:)
p => a(9:1:-2,1:9:3)
call f(p)
.
.
.
```

The descriptor for actual argument p would contain the following values:

- The first longword (bytes 0 to 3) contains the base address (assigned at run-time).
- The second longword (bytes 4 to 7) is set to 4 (size of a single element).

- The third longword (bytes 8 to 11) contains the A0 offset of -112.
- The fourth longword (bytes 12 to 15) contains 3 (array is defined and deallocation is not allowed).
- The fifth longword (bytes 16 to 19) contains 2 (rank).
- The sixth longword (bytes 20-23) is reserved.
- The seventh, eighth, and ninth longwords (bytes 24 to 35) contain information for the first dimension, as follows:
  - 5 (extent)
  - -8 (distance between elements)
  - 9 (the lower bound)
- For the second dimension, the tenth, eleventh, and twelfth longwords (bytes 36 to 47) contain:
  - 3 (extent)
  - 120 (distance between elements)
  - 1 (the lower bound)
- Byte 47 is the last byte for this example.

NOTE. The format for the descriptor on systems using Intel® 64 architecture and those using IA-64 architecture is identical to that on systems using IA-32 architecture, except that all fields are 8-bytes long, instead of 4-bytes.

Handling Large Arrays

When compiling a program with an array greater than 2GB on systems using Intel® 64 architecture and running Linux OS, you may need to specify certain compiler options. Specifically, you may need to use either the `-mcmodel=medium` or `-mcmodel=large` compiler option and also the `-shared-intel` option. For more information, see Specifying Memory Models to use with Systems Based on Intel® 64 Architecture.

Handling Character Strings

By default, Intel® Fortran passes a hidden length argument for strings. The hidden length argument consists of an unsigned 4-byte integer (for systems based on IA-32 architecture) or unsigned 8-byte integer (for systems based on Intel® 64 architecture and those based on IA-64 architecture), always passed by value, added to the end of the argument list. You can alter the default way strings are passed by using attributes. The following table shows the effect of various attributes on passed strings.

Table 42: Effect of ATTRIBUTES Options on Character Strings Passed as Arguments

Argument	Default	C	C, REFERENCE	STDCALL (Windows* OS)	STDCALL, REFERENCE (Windows* OS)
String	Passed by reference, along with length	First character converted to INTEGER(4) and passed by value	Passed by reference, along with length	First character converted to INTEGER(4) and passed by value	Passed by reference, along with length
String with VALUE option	Error	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value	First character converted to INTEGER(4) and passed by value
String with REFERENCE option	Passed by reference, possibly along with length	Passed by reference, no length	Passed by reference, no length	Passed by reference, no length	Passed by reference, no length

The important things to note about the above table are:

- Character strings without the VALUE or REFERENCE attribute that are passed to C or STDCALL routines are not passed by reference. Instead, only the first character is passed and it is passed by value.
- Character strings with the VALUE option passed to C or STDCALL routines are not passed by reference. Instead, only the value of the first character is passed.

- For string arguments with default ATTRIBUTES, ATTRIBUTES C, REFERENCE, or ATTRIBUTES STDCALL, REFERENCE:
  - When -nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg (Windows OS) is set, the length of the string is pushed (by value) on the stack after all of the other arguments. This is the default.
  - When -mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg (Windows OS) is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.
- For string arguments passed by reference with default ATTRIBUTES:
  - When -nomixed-str-len-arg (Linux OS and Mac OS X) or /iface:nomixed_str_len_arg is set, the length of the string is not available to the called procedure. This is the default.
  - When -mixed-str-len-arg (Linux OS and Mac OS X) or /iface:mixed_str_len_arg is set, the length of the string is pushed (by value) on the stack immediately after the address of the beginning of the string.

Since all strings in C are pointers, C expects strings to be passed by reference, without a string length. In addition, C strings are null-terminated while Fortran strings are not. There are two basic ways to pass strings between Fortran and C: convert Fortran strings to C strings, or write C routines to accept Fortran strings.

To convert a Fortran string to C, choose a combination of attributes that passes the string by reference without length, and null terminate your strings. For example:

```
INTERFACE
  SUBROUTINE Pass_Str (string)
    !DEC$ ATTRIBUTES C, DECORATE,ALIAS:'Pass_Str' :: Pass_Str
    CHARACTER*(*) string
    !DEC$ ATTRIBUTES REFERENCE :: string
  END SUBROUTINE
END INTERFACE
CHARACTER(40) forstring
DATA forstring /'This is a null-terminated string.'C/
```

The following example shows the extension of using the null-terminator for the string in the Fortran DATA statement:

```
DATA forstring /'This is a null-terminated string.'C/
```

The C interface is:

```
void Pass_Str (char *string)
```

To get your C routines to accept Fortran strings, C must account for the length argument passed along with the string address. For example:

```
! Fortran code
INTERFACE
  SUBROUTINE Pass_Str (string)
  CHARACTER*(*) string
END INTERFACE
```

The C routine must expect two arguments:

```
void pass_str (char *string, unsigned int length_arg )
```

This interface handles the hidden-length argument, but you must still reconcile C strings that are null-terminated and Fortran strings that are not. In addition, if the data assigned to the Fortran string is less than the declared length, the Fortran string will be blank padded.

Rather than trying to handle these string differences in your C routines, the best approach in Fortran/C mixed programming is to adopt C string behavior whenever possible. An added benefit for using C strings on Windows* operating systems is that Windows API routines and most C library functions expect null-terminated strings.

Fortran functions that return a character string using the syntax `CHARACTER*(*)` place a hidden string argument and the length of the string at the beginning of the argument list.

C functions that implement such a Fortran function call must declare this hidden string argument explicitly and use it to return a value. The C return type should be void. However, you are more likely to avoid errors by not using character-string return functions. Use subroutines or place the strings into modules or global variables whenever possible.

Handling Character Strings in Visual Basic and MASM

The following information on Visual Basic and MASM applies to Windows operating systems only.

Visual Basic strings must be passed by value to Fortran. Visual Basic strings are actually stored as structures containing length and location information. Passing by value dereferences the structure and passes just the string location, as Fortran expects. For example:

```
! In Basic
 Declare Sub forstr  Lib "forstr.dll" (ByVal Bstring as String)
 DIM bstring As String * 40 Fixed-length string
 CALL forstr(bstring)
! End Basic code
! In Fortran
 SUBROUTINE forstr(s)
 !DEC$ ATTRIBUTES STDCALL :: forstr
 !DEC$ ATTRIBUTES REFERENCE :: s
```

```
CHARACTER(40) s
s = 'Hello, Visual Basic!'
END
```

The Fortran directive !DEC$ ATTRIBUTES STDCALL and the ATTRIBUTES REFERENCE property on variable arguments together inform Fortran not to expect the hidden length arguments to be passed from the Visual Basic calling program. The name in the Visual Basic program is specified as lowercase since STDCALL makes the Fortran name lowercase.

MASM does not add either a string length or a null character to strings by default. To append the string length, use the syntax:

lenstring BYTE "String with length", LENGTHOF lenstring

To add a null character, append it by hand to the string:

```
nullstring BYTE "Null-terminated string", 0
```

Returning Character Data Types

If a Fortran program expects a function to return data of type CHARACTER, the Fortran compiler adds two additional arguments to the beginning of the called procedure's argument list:

- The first argument is a pointer to the location where the called procedure should store the result.
- The second is the maximum number of characters that must be returned, padded with blank spaces if necessary.

The called routine must copy its result through the address specified in the first argument. The following example shows the Fortran code for a return character function called MAKECHARS and a corresponding C routine.

Example of Returning Character Types from C to Fortran

Fortran code

```
CHARACTER*10 CHARS, MAKECHARS
DOUBLE PRECISION X, Y
CHARS = MAKECHARS( X, Y )
```

Corresponding C Routine

```
void MAKECHARS ( result, length, x, y );
char *result;
int length;
double *x, *y;
{
...program text, producing returnvalue...
for (i = 0; i < length; i++ ) {
```

```
result[i] = returnvalue[i];
}
}
```

In the above example, the following restrictions and behaviors apply:

- The function's length and result do not appear in the call statement; they are added by the compiler.
- The called routine must copy the result string into the location specified by `result`; it must not copy more than `length` characters.
- If fewer than `length` characters are returned, the return location should be padded on the right with blanks; Fortran does not use zeros to terminate strings.
- The called procedure is type `void`.
- On Windows, you must use uppercase names for C routines or Microsoft attributes and `INTERFACE` blocks to make the calls using lower case.

Handling User-Defined Types

Fortran 95/90 supports user-defined types (data structures similar to C structures). User-defined types can be passed in modules and common blocks just as other data types, but the other language must know the type's structure. For example:

Fortran Code:

```
TYPE LOTTA_DATA
   SEQUENCE
   REAL A
   INTEGER B
   CHARACTER(30) INFO
   COMPLEX CX
   CHARACTER(80) MOREINFO
END TYPE LOTTA_DATA
TYPE (LOTTA_DATA) D1, D2
COMMON /T_BLOCK/ D1, D2
```

In the Fortran code above, the SEQUENCE statement preserves the storage order of the derived-type definition.

C Code:

```
/* C code accessing D1 and D2 */
extern struct {
  struct {
     float a;
     int b;
     char info[30];
```

```
      struct {
         float real, imag;
         } cx;
      char moreinfo[80];
   } d1, d2;
} T_BLOCK;
```

Intel(R) Fortran/Visual Basic* Mixed-Language Programs

Interoperability with C

It is often desirable to have a program which contains both Fortran and C code, and in which routines written in one language are able to call routines written in the other. The Intel® Fortran compiler supports the Fortran 2003 standardized mechanism for allowing Fortran code to reliably communicate (or *interoperate*) with C code. The following describes interoperability requirements for types, variables, and procedures.

Interoperability of Intrinsic Types

The intrinsic module ISO_C_BINDING contains named constants that hold kind type parameter values for intrinsic types.

The more commonly used types are included in the following table. The following applies:

- Integer types in Fortran are always signed. In C, integer types may be specified as signed or unsigned, but are unsigned by default.
- The values of C_LONG, C_SIZE_T, C_LONG_DOUBLE, AND C_LONG_DOUBLE_COMPLEX are different on different platforms.

Named constant from ISO_C_BINDING (kind type parameter if value is positive)	**C type**	**Equivalent Fortran type**
C_SHORT	short int	INTEGER(KIND=2)
C_INT	int	INTEGER(KIND=4)
C_LONG	long int	INTEGER (KIND=4 or 8)
C_LONG_LONG	long long int	INTEGER(KIND=8)

C_SIGNED_CHAR	signed char unsigned char	INTEGER(KIND=1)
C_SIZE_T	size_t	INTEGER(KIND=4 or 8)
C_INT8_T	int8_t	INTEGER(KIND=1)
C_INT16_T	int16_t	INTEGER(KIND=2)
C_INT32_T	int32_t	INTEGER(KIND=4)
C_INT64_T	int64_t	INTEGER(KIND=8)
C_FLOAT	float	REAL(KIND=4)
C_DOUBLE	double	REAL(KIND=8)
C_LONG_DOUBLE	long double	REAL(KIND=8 or 16)
C_FLOAT_COMPLEX	float _Complex	COMPLEX(KIND=4)
C_DOUBLE_COMPLEX	double _Complex	COMPLEX(KIND=8)
C_LONG_DOUBLE_COMPLEX	long double _Complex	COMPLEX(KIND=8 or 16)
C_BOOL	_Bool	LOGICAL(KIND=1)
C_CHAR	char	CHARACTER(LEN=1)

While there are named constants for all possible C types, every type is not necessarily supported on every processor. Lack of support is indicated by a negative value for the constant in the module.

For a character type to be interoperable, you must either omit the length type parameter or specify it using an initialization expression whose value is one.

Interoperability with C Pointers

For interoperating with C pointers, the module ISO_C_BINDING contains the derived types C_PTR and C_FUNPTR, which are interoperable with C object and function type pointers, respectively.

These types, as well as certain procedures in the module, provide the mechanism for passing dynamic arrays between the two languages. Because its elements need not be contiguous in memory, a Fortran pointer target or assumed-shape array cannot be passed to C. However, you can pass an allocated allocatable array to C, and you can associate an array allocated in C with a Fortran pointer.

Interoperability of Derived Types

For a derived type to be interoperable with C, you must specify the BIND(C) attribute:

```
TYPE, BIND(C) :: MYTYPE
```

Additionally, as shown in the examples that follow, each component must have an interoperable type and interoperable type parameters, must not be a pointer, and must not be allocatable. This allows Fortran and C types to correspond.

```
typedef struct {
int m, n;
float r;
} myctype
```

The above is interoperable with the following:

```
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
INTEGER(C_INT) :: I, J
REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

Interoperability of Variables

A scalar Fortran variable is interoperable if its type and type parameters are interoperable and it is not a pointer.

An array Fortran variable is interoperable if its type and type parameters are interoperable and it has an explicit shape or assumed size. It interoperates with a C array of the same type, type parameters, and shape, but with subscripts reversed.

For example, a Fortran array declared as `INTEGER :: A(18, 3:7, *)` is interoperable with a C array declared as `int b[][5][18]`.

Interoperability of Procedures

For a procedure to be interoperable, it must have an explicit interface and be declared with the BIND attribute, as shown in the following:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C)
```

In the case of a function, the result must be scalar and interoperable.

A procedure has an associated binding label, which is global in scope. This label is the name by which the C processor knows it and is, by default, the lower-case version of the Fortran name. For example, the above function has the binding label func. You can specify an alternative binding label as follows:

```
FUNCTION FUNC(I, J, K, L, M), BIND(C, NAME='myC_Func')
```

All dummy arguments must be interoperable. Furthermore, you must ensure that either the Fortran routine uses the VALUE attribute for scalar dummy arguments, or that the C routine receives these scalar arguments as pointers to the scalar values. Consider the following call to this C function:

```
intc_func(int x, int *y);
```

As shown here, the interface for the Fortran call to c_func must have x passed with the VALUE attribute, but y should not have the VALUE attribute, since it is received as a pointer:

```
INTERFACE
INTEGER (C_INT) FUNCTION C_FUNC(X, Y) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER (C_INT), VALUE :: X
INTEGER (C_INT) :: Y
END FUNCTION C_FUNC
END INTERFACE
```

Alternatively, the declaration for y can be specified as a C_PTR passed by value:

```
TYPE (C_PTR), VALUE :: Y
```

To pass a scalar Fortran variable of type character, the character length must be one.

Interoperability of Global Data

A module variable or a common block can interoperate with a C global variable if the Fortran entity uses the BIND attribute and the members of that entity are also interoperable. For example, consider the entities C_EXTERN, C2, COM and SINGLE in the following module:

```
MODULE LINK_TO_C_VARS
USE, INTRINSIC :: ISO_C_BINDING
INTEGER(C_INT), BIND(C) :: C_EXTERN
INTEGER(C_LONG) :: C2
BIND(C, NAME='myVariable') :: C2
COMMON /COM/ R,S
REAL(C_FLOAT) :: R,S,T
BIND(C) :: /COM/, /SINGLE/
COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS
```

These can interoperate with the following C external variables:

```
int c_extern;
long myVariable;
struct {float r, s;} com;
float single;
```

Example of Fortran Calling C

The following example calls a C function.

C Function Prototype:

```
int C_Library_Function(void* sendbuf, int sendcount, int *recvcounts);
```

Fortran Modules:

```
MODULE FTN_C_1
USE, INTRINSIC :: ISO_C_BINDING
END MODULE FTN_C_1

MODULE FTN_C_2
INTERFACE
INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
(SENDBUF, SENDCOUNT, RECVCOUNTS) &
BIND(C, NAME='C_Library_Function')
USE FTN_C_1
IMPLICIT NONE
TYPE (C_PTR), VALUE :: SENDBUF
INTEGER (C_INT), VALUE :: SENDCOUNT
TYPE (C_PTR), VALUE :: RECVCOUNTS
END FUNCTION C_LIBRARY_FUNCTION
END INTERFACE
END MODULE FTN_C_2
```

Fortran Calling Sequence:

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
USE FTN_C_2
...
REAL (C_FLOAT), TARGET :: SEND(100)
INTEGER (C_INT) :: SENDCOUNT
INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
...
ALLOCATE( RECVCOUNTS(100) )
...
CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
C_LOC(RECVCOUNTS))
...
```

Example of C Calling Fortran

The following example calls a Fortran subroutine called SIMULATION. This subroutine corresponds to the C void function simulation.

Fortran Code:

```
SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS) BIND(C)
USE, INTRINSIC :: ISO_C_BINDING
IMPLICIT NONE
INTEGER (C_LONG), VALUE :: ALPHA
REAL (C_DOUBLE), INTENT(INOUT) :: BETA
INTEGER (C_LONG), INTENT(OUT) :: GAMMA
REAL (C_DOUBLE),DIMENSION(*),INTENT(IN) :: DELTA
TYPE, BIND(C) :: PASS
INTEGER (C_INT) :: LENC, LENF
TYPE (C_PTR) :: C, F
END TYPE PASS
TYPE (PASS), INTENT(INOUT) :: ARRAYS
REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
REAL (C_FLOAT), POINTER :: C_ARRAY(:)
...
! Associate C_ARRAY with an array allocated in C
CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/) )
...
! Allocate an array and make it available in C
ARRAYS%LENF = 100
ALLOCATE (ETA(ARRAYS%LENF))
ARRAYS%F = C_LOC(ETA)
...
END SUBROUTINE SIMULATION
```

C Struct Declaration

```
struct pass {int lenc, lenf; float *c, *f;};
```

C Function Prototype:

```
void simulation(long alpha, double *beta, long *gamma, double delta[], struct pass *arrays);
```

C Calling Sequence:

```
simulation(alpha, &beta, &gamma, delta, &arrays);
```

Compiling and Linking Intel® Fortran/C Programs

Your application can contain both C and Fortran source files. If your main program is a Fortran source file (myprog.for) that calls a routine written in C (cfunc.c), you can use the following sequence of commands to build your application.

Linux* OS and Mac OS* X:

```
icc -c cfunc.c
ifort -o myprog myprog.for cfunc.o
```

Windows* OS:

```
icl /c cfunc.c
ifort myprog.for cfunc.obj
 /link /out:myprog.exe
```

The `icc` or `icl` command for Intel® C++ or the `cl` command (for Microsoft Visual C++*) compiles `cfunc.c`. The `-c` or `/c` option specifies that the linker is not called. This command creates `cfunc.o` (Linux OS and Mac OS X) or `cfunc.obj` (Windows OS).

The ifort command compiles `myprog.for` and links `cfunc.o` (Linux OS and Mac OS X) or `cfunc.obj` (Windows OS) with the object file created from `myprog.for` to create the executable.

Additionally, on Linux OS and Mac OS X, you may need to specify one or more of the following options:

- Use the -cxxlib compiler option to tell the compiler to link using the C++ run-time libraries provided by gcc. By default, C++ libraries are not linked with Fortran applications.
- Use the `-fexceptions` compiler option to enable C++ exception handling table generation so C++ programs can handle C++ exceptions when there are calls to Fortran routines on the call stack. This option causes additional information to be added to the object file that is required during C++ exception handling. By default, mixed Fortran/C++ applications abort in the Fortran code if a C++ exception is thrown.
- Use the `-nofor_main` compiler option if your C/C++ program calls an Intel Fortran subprogram, as shown:

```
icc -c  cmain.c
ifort -nofor_main cmain.o fsub.f90
```

Calling C Procedures from an Intel® Fortran Program

Naming Conventions

By default, the Fortran compiler converts function and subprogram names to lower case for Linux OS and Mac OS X and upper case for Windows OS. The C compiler never performs case conversion. A C procedure called from a Fortran program must, therefore, be named using the appropriate case. For example, consider the following calls:

`CALL PROCNAME()`	The C procedure must be named `PROCNAME`.

`X=FNNAME()`	The C procedure must be named `FNNAME`

In the first call, any value returned by `PROCNAME` is ignored. In the second call to a function, `FNNAME` must return a value.

Passing Arguments Between Fortran and C Procedures

By default, Fortran subprograms pass arguments by reference; that is, they pass a pointer to each actual argument rather than the value of the argument. C programs, however, pass arguments by value. Consider the following:

- When a Fortran program calls a C function, the C function's formal arguments must be declared as pointers to the appropriate data type.
- When a C program calls a Fortran subprogram, each actual argument must be specified explicitly as a pointer.

For Windows systems using IA-32 architecture only, you can alter the default calling convention. You can use either the `/iface:stdcall` option (`stdcall`) or the `/iface:cvf` option (Compaq* and Powerstation compatibility) to change the default calling convention, or the `VALUE` or C attributes in an explicit interface using the ATTRIBUTES directive. For more information on the ATTRIBUTES directive, see the Intel® Fortran Language Reference.

Both options cause the routine compiled and routines that it calls to have a @<*n*> appended to the external symbol name, where n is the number of bytes of all parameters. Both options assume that any routine called from a Fortran routine compiled this way will do its own stack cleanup, "callee pops." `/iface:cvf` also changes the way that `CHARACTER` variables are passed. With `/iface:cvf`, `CHARACTER` variables are passed as address/length pairs (that is, `/iface:mixed_str_len_arg`).

Using Libraries

14

Supplied Libraries

Libraries are simply an indexed collection of object files that are included as needed in a linked program. Combining object files into a library makes it easy to distribute your code without disclosing the source. It also reduces the number of command-line entries needed to compile your project.

Intel® Fortran provides different types of libraries, such as static or DLL, single-threaded or multi-threaded.

On Linux* OS and Mac OS* X systems, you can use the `-shared-intel` compiler option on the command line to specify that you want to use the dynamic versions of all Intel libraries.

The tables below show the libraries provided for the compiler. Except where noted, listed libraries apply to systems based on IA-32 architecture, systems based on Intel® 64 architecture and systems based on IA-64 architecture.

The run-time libraries have associated message catalog files, described in Run-Time Library Message Catalog Location.

The file `fredist.txt` in the `<install-dir> /Documentation` folder lists the Intel compiler libraries that are redistributable.

Table 45: Libraries provided on Windows* OS systems:

File	Description
`ifauto.lib`	Fortran interfaces to Automation objects
`ifcom.lib`	Fortran interfaces to COM support
`ifconsol.lib`	QuickWin stub support
`ifdlg100.dll`	Provides ActiveX* control support to the dialog procedures
`iflogm.lib`	Dialog support
`ifmodintr.lib`	Intrinsic module support
`ifqw_mdi.lib`	QuickWin multi-document support library
`ifqw_sdi.lib`	QuickWin single document support library

File	Description
ifqwin.lib	QuickWin support library
ifwin.lib	Miscellaneous Windows support
libguide.lib	OpenMP* static library for the parallelizer tool
libguide40.lib libguide40.dll	These two libraries make up a dynamic library for the parallelizer tool
libguide_stats.lib	OpenMP static library for the parallelizer tool with performance statistics and profile information
libguide40_stats.lib libguide40_stats.dll	These two libraries make up a dynamic library for the parallelizer tool with performance statistics and profile information
libifcore.lib	Intel-specific Fortran I/O intrinsic support library
libifcoremd.lib libifcoremd.dll	...when compiled with /MD
libifcoremdd.lib libifcoremdd.dll	...when compiled with /MDd
libifcoremt.lib	...when compiled with /MT
libifcorert.lib libifcorert.dll	...when compiled with /MDs
libifcorertd.lib libifcorertd.dll	...when compiled with /MDsd
libifport.lib	Portability, POSIX*, and NLS* support library
libifportmd.dll libifportmd.lib	...when compiled with /MD
libirc.lib	Intel-specific library (optimizations)

File	Description
libircmt.lib	Multithreaded Intel-specific library (optimizations)
libm.lib	Math library
libmmd.lib libmmd.dll	These two libraries make up a dynamic library for the multithreaded math library used when compiling with /MD
libmmdd.lib libmmdd.dll	These two libraries make up a debug dynamic library for the multithreaded math library used when compiling with /MD
libmmds.lib	Static math library built multithread
libmmt.lib	Multithreaded math library used when compiling with /MT
libompstub.lib	Library that resolves references to OpenMP* subroutines when OpenMP is not in use
svml_disp.lib	Short-vector math library (used by vectorizer). Not provided on systems based on IA-64 architecture.
svml_dispmt.lib	Multithread short-vector math library (used by vectorizer). Not provided on systems based on IA-64 architecture.

Table 46: Libraries provided on Linux* OS and Mac OS* X systems:

File	Description
for_main.o	main routine for Fortran programs
libcxaguard.a libcxaguard.so (.dylib for Mac OS X) libcxaguard.so.5 (Linux IA-32 and Intel® 64 architectures) libcxaguard.so.6 (Linux IA-64 architecture)	Used for interoperability with the -cxxlib option.
libguide.a	OpenMP* static library for the parallelizer tool

File	Description
libguide.so (.dylib for Mac OS X)	
libguide_stats.a libguide_stats.so (.dylib for Mac OS X)	Support for parallelizer tool with performance and profile information
libifcore.a libifcore.so (.dylib for Mac OS X) libifcore.so.5 (Linux OS IA-32 and Intel® 64 architectures) libifcore.so.6 (IA-64 architecture)	Intel-specific Fortran run-time library
libifcore_pic.a libifcoremt_pic.a	Intel-specific Fortran static libraries; Linux OS only. These support position independent code and allow creation of shared libraries linked to Intel-specific Fortran *static* run-time libraries, instead of shared run-time libraries.
libifcoremt.a libifcoremt.so (.dylib for Mac OS X) libifcoremt.so.5 (Linux OS IA-32 and Intel® 64 architectures) libifcoremt.so.6 (IA-64 architecture)	Multithreaded Intel-specific Fortran run-time library
libifport.a libifport.so (.dylib for Mac OS X)	Portability and POSIX support

File	Description
`libifport.so.5` (Linux OS IA-32 and Intel® 64 architectures)	
`libifport.so.6` (IA-64 architecture)	
`libifportmt.dylib` (Mac OS X only)	
`libimf.a`	Math library
`libimf.so` (`.dylib` for Mac OS X)	
`libirc.a`	Intel-specific library (optimizations)
`libirc_s.a`	
`libirc.dylib` (Mac OS X)	
`libintlc.so` (`.dylib` for Mac OS X)	Dynamic versions of libirc
`libompstub.a`	Library that resolves references to OMP subroutines when OMP is not in use
`libsvml.a`	Short vector math library
`libsvml.dylib` (Mac OS X)	

Creating Static Libraries

Executables generated using static libraries are no different than executables generated from individual source or object files. Static libraries are not required at runtime, so you do not need to include them when you distribute your executable. At compile time, linking to a static library is generally faster than linking to individual source files.

When compiling a static library from the ifort command line, include the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) compiler option to suppress linking. Without this option, the compiler generates an error because the library does not contain a main program.

To build a static library (Linux OS):

1. Use the -c option to generate object files from the source files:
   ```
   ifort -c my_source1.f90 my_source2.f90 my_source3.f90
   ```
2. Use the GNU ar tool to create the library file from the object files:
   ```
   ar rc my_lib.a my_source1.o my_source2.o my_source3.o
   ```
3. Compile and link your project with your new library:
   ```
   ifort main.f90 my_lib.a
   ```

If your library file and source files are in different directories, use the `-Ldir` option to indicate where your library is located:

```
ifort -L/for/libs main.f90 my_lib.a
```

To build a static library (Mac OS X):

1. Use the following command line to generate object files and create the library file:
   ```
   ifort -o my_lib.a -staticlib mysource1.f90 mysource2.f90 mysource3.f90
   ```
2. Compile and link your project with your new library:
   ```
   ifort main.f90 my_lib.a
   ```

If your library file and source files are in different directories, use the `-Ldir` option to indicate where your library is located:

```
ifort -L/for/libs main.f90 my_lib.a
```

To build a static library (Windows OS):

To build a static library from the integrated development environment (IDE), select the Fortran Static Library project type.

To build a static library using the command line:

1. Use the /c option to generate object files from the source files:
   ```
   ifort /c my_source1.f90 my_source2.f90
   ```
2. Use the Microsoft LIB tool to create the library file from the object files:
   ```
   lib /out:my_lib.lib my_source1.obj my_source2.obj
   ```
3. Compile and link your project with your new library:
   ```
   ifort main.f90 my_lib.lib
   ```

Creating Shared Libraries

Shared libraries, also referred to as dynamic libraries, are linked differently than static libraries. At compile time, the linker insures that all the necessary symbols are either linked into the executable, or can be linked at runtime from the shared library. Executables compiled from shared libraries are smaller, but the shared libraries must be included with the executable to function correctly. When multiple programs use the same shared library, only one copy of the library is required in memory.

To create a shared library from a Fortran source file, process the files using the ifort command:

- You must specify the `-shared` option (Linux* OS) or the `-dynamiclib` option (Mac OS* X) to create the `.so` or `.dylib` file. On Linux OS and Mac OS X operating systems using either IA-32 architecture or Intel® 64 architecture, you must also specify `-fpic` for the compilation of each object file you want to include in the shared library.
- You can specify the `-o` *`output`* option to name the output file.
- If you omit the `-c` option, you will create a shared library (`.so` file) directly from the command line in a single step.
- If you also omit the `-o` *`output`* option, the file name of the first Fortran file on the command line is used to create the file name of the `.so` file. You can specify additional options associated with shared library creation.
- If you specify the `-c` option, you will create an object file (`.o` file) that you can name with the `-o` option. To create a shared library, process the `.o` file with `ld` , specifying certain options associated with shared library creation.

Creating a Shared Library

There are several ways to create a shared library.

You can create a shared library file with a single ifort command:

```
ifort -shared -fpic octagon.f90 (Linux OS)
ifort -dynamiclib octagon.f90 (Mac OS* X)
```

The `-shared` or `-dynamiclib` option is required to create a shared library. The name of the source file is `octagon.f90`. You can specify multiple source files and object files.

The `-o` option was omitted, so the name of the shared library file is `octagon.so` (Linux OS) or`octagon.dylib` (Mac OS X).

You can use the `-static-intel` option to force the linker to use the static versions of the Intel-supplied libraries.

You can also create a shared library file with a combination of ifort and ld (Linux OS) or libtool (Mac OS X) commands:

First, create the `.o` file, such as `octagon.o` in the following example:

```
ifort -c -fpic octagon.f90
```

The file `octagon.o` is then used as input to the `ld` (Linux OS) or `libtool` (Mac OS X) command to create the shared library. The following example shows the command to create a shared library named `octagon.so` on a Linux operating system:

```
ld -shared octagon.o \
     -lifport -lifcoremt -limf -lm -lcxa \
     -lpthread -lirc -lunwind -lc -lirc_s
```

Note the following:

- When you use `ld`, you need to list all Fortran libraries. It is easier and safer to use the ifort command. On Mac OS X, you would use libtool.
- The `-shared` option is required to create a shared library. On Mac OS X, use the `-dynamiclib` option, and also specify the following: `-arch_only i386`, `-noall_load`, `-weak_references_mismatches non-weak`.
- The name of the object file is `octagon.o`. You can specify multiple object (`.o`) files.
- The `-lifport` option and subsequent options are the standard list of libraries that the ifort command would have otherwise passed to `ld` or `libtool`. When you create a shared library, all symbols must be resolved.

It is probably a good idea to look at the output of the `-dryrun` command to find the names of all the libraries used so you can specify them correctly.

If you are using the `ifort` command to link, you can use the `-Qoption` command to pass options to the ld linker. (You cannot use `-Qoption` on the `ld` command line.)

For more information on relevant compiler options, see the Compiler Options reference.

See also the `ld(1)` reference page.

Shared Library Restrictions

When creating a shared library with `ld`, be aware of the following restrictions:

- Shared libraries must not be linked with archive libraries.

When creating a shared library, you can only depend on other shared libraries for resolving external references. If you need to reference a routine that currently resides in an archive library, either put that routine in a separate shared library or include it in the shared library being created. You can specify multiple object (`.o`) files when creating a shared library.

To put a routine in a separate shared library, obtain the source or object file for that routine, recompile if necessary, and create a separate shared library. You can specify an object file when recompiling with the `ifort` command or when creating the shared library with the `ld` command.

To include a routine in the shared library being created, put the routine (source or object file) with other source files that make up the shared library and recompile if necessary.

Now create the shared library, making sure that you specify the file containing that routine either during recompilation or when creating the shared library. You can specify an object file when recompiling with the `ifort` command or when creating the shared library with the `ld` or libtool command.

- When creating shared libraries, all symbols must be defined (resolved).

  Because all symbols must be defined to `ld` when you create a shared library, you must specify the shared libraries on the `ld` command line, including all standard Intel Fortran libraries. The list of standard Intel Fortran libraries can be specified by using the `-l`*`string`* option.

Installing Shared Libraries

Once the shared library is created, it must be installed for private or system-wide use before you run a program that refers to it:

- To install a *private* shared library (when you are testing, for example), set the environment variable LD_LIBRARY_PATH, as described in ld`(1)`. For Mac OS X, set the environment variable `DYLD_LIBRARY_PATH`.
- To install a *system-wide* shared library, place the shared library file in one of the standard directory paths used by `ld` or `libtool`.

Calling Library Routines

The following table shows the groups of Intel Fortran library routines and the USE statement required to include the interface definitions for the routines in that group:

Routines	USE statement
Portability	USE IFPORT
POSIX*	USE IFPOSIX
Miscellaneous Run-Time	USE IFCORE
The following are Windows only:	
Automation (AUTO) (systems using IA-32 architecture only)	USE IFAUTO
Component Object Model (COM) (systems using IA-32 architecture only)	USE IFCOM
Dialog (systems using IA-32 architecture only)	USE IFLOGM
Graphics	USE IFQWIN
National Language Support	USE IFNLS
QuickWin	USE IFQWIN
Serial port I/O (SPORT)(systems using IA-32 architecture only)	USE IFPORT

Module Routines lists topics that provide an overview of the different groups of library routines as well as calling syntax for the routines. For example, add the following USE statement (before any data declaration statements, such as IMPLICIT NONE or INTEGER):

```
USE IFPORT
```

If you want to minimize compile time for source files that use the Intel Fortran library routines, add the ONLY keyword to the USE statement. For example:

```
USE IFPORT, only: getenv
```

Using the ONLY keyword limits the number of interfaces for that group of library routines.

To view the actual interface definitions, view the `.f90` file that corresponds to the `.mod` file. For example, if a routine requires a USE IFCORE, locate and use a text editor to view the file `ifcore.f90` in the standard INCLUDE directory.

You should avoid copying the actual interface definitions contained in the `ifport.f90` (or `ifcore.f90`, ...) into your program because future versions of Intel Fortran might change these interface definitions.

Similarly, some of the library interface `.f90` files contain USE statements for a subgrouping of routines. However, if you specify a USE statement for such a subgroup, this module name may change in future version of Intel Fortran. Although this will make compilation times faster, it might not be compatible with future versions of Intel Fortran.

Portability Considerations

Portability Library Overview

Intel® Fortran includes functions and subroutines that ease porting of code to or from a PC, or allow you to write code on a PC that is compatible with other platforms.

The portability library is called `LIBIFPORT.LIB` (Windows* OS) or `libifport.a` (Linux* OS and Mac OS* X). Frequently used functions are included in a portability module called IFPORT.

The portability library also contains IEEE* POSIX library functions. These functions are included in a module called IFPOSIX.

You can use the portability library in one of two ways:

- Add the statement USE **IFPORT** to your program. This statement includes the IFPORT module.
- Call portability routines using the correct parameters and return value.

The portability library is passed to the linker by default during linking. To prevent this, specify the `-fpscomp nolibs` (Linux OS and Mac OS X) or `/fpscomp:nolibs` (Windows OS) option.

Using the IFPORT mod file provides interface blocks and parameter definitions for the routines, as well as compiler verification of calls.

See also:

- Using the IFPORT Portability Module
- Portability Routines

Using the IFPORT Portability Module

Using the IFPORT module provides interface blocks and parameter definitions for the portability routines, as well as compiler verification of calls.

Some routines in this library can be called with different sets of arguments, and sometimes even as a function instead of a subroutine. In these cases, the arguments and calling mechanism determine the meaning of the routine. The IFPORT module contains generic interface blocks that give procedure definitions for these routines.

Fortran 95/90 contains intrinsic procedures for many of the portability functions. The portability routines are extensions to the Fortran 95 standard. When writing new code, use Fortran 95/90 intrinsic procedures whenever possible (for portability and performance reasons).

Portability Routines

This section describes some of the portability routines and how to use them.

Refer to the Portability Routines table as you read through this topic.

Information Retrieval Routines

Information retrieval procedures return information about system commands, command-line arguments, environment variables, and process or user information.

Group, user, and process ID are INTEGER(4) variables. Login name and host name are character variables. The functions GETGID and GETUID are provided for portability, but always return 1.

Process Control Routines

Process control routines control the operation of a process or subprocess. You can wait for a subprocess to complete with either SLEEP or ALARM, monitor its progress and send signals via KILL, and stop its execution with ABORT.

In spite of its name, KILL does not necessarily stop execution of a program. Rather, the routine signaled could include a handler routine that examines the signal and takes appropriate action depending on the code passed.

Note that when you use SYSTEM, commands are run in a separate shell. Defaults set with the SYSTEM function, such as current working directory or environment variables, do not affect the environment the calling program runs in.

The portability library does not include the FORK routine. On Linux* OS and Mac OS* X systems, FORK creates a duplicate image of the parent process. Child and parent processes each have their own copies of resources, and become independent from one another. On Windows* OS systems, you can create a child process (called a thread), but both parent and child processes share the same address space and share system resources.

Numeric Values and Conversion Routines

Numeric values and conversion routines are available for calculating Bessel functions, data type conversion, and generating random numbers. Some of these functions have equivalents in standard Fortran 95/90, in which case the standard Fortran routines should be used.

Data object conversion can be accomplished by using the INT intrinsic function instead of LONG or SHORT. The intrinsic subroutines RANDOM_NUMBER and RANDOM_SEED perform the same functions as the random number functions listed in the table showing numeric values and conversion routines.

Other bit manipulation functions such as AND, XOR, OR, LSHIFT, and RSHIFT are intrinsic functions. You do not need the IFPORT module to access them. Standard Fortran 95/90 includes many bit operation routines, which are listed in the Bit Operation and Representation Routines table.

Input and Output Routines

The portability library contains routines that change file properties, read and write characters and buffers, and change the offset position in a file. These input and output routines can be used with standard Fortran input or output statements such as READ or WRITE on the same files, provided that you take into account the following:

- When used with direct files, after an FSEEK, GETC, or PUTC operation, the record number is the number of the next whole record. Any subsequent normal Fortran I/O to that unit occurs at the next whole record. For example, if you seek to absolute location 1 of a file whose record length is 10, the NEXTREC returned by an INQUIRE would be 2. If you seek to absolute location 10, NEXTREC would still return 2.
- On units with CARRIAGECONTROL='FORTRAN' (the default), PUTC and FPUTC characters are treated as carriage control characters if they appear in column 1.
- On sequentially formatted units, the C string "\n"c, which represents the carriage return/line feed escape sequence, is written as CHAR(13) (carriage return) and CHAR(10) (line feed), instead of just line feed, or CHAR(10). On input, the sequence 13 followed by 10 is returned as just 10. (The length of character string "\n"c is 1 character, whose ASCII value, indicated by ICHAR('\n'c), is 10.)
- Reading and writing is in a raw form for direct files. Separators between records can be read and overwritten. Therefore, be careful if you continue using the file as a direct file.

I/O errors arising from the use of these routines result in an Intel Fortran run-time error.

Some portability file I/O routines have equivalents in standard Fortran 95/90. For example, you could use the ACCESS function to check a file specified by name for accessibility according to mode. It tests a file for read, write, or execute permission, as well as checking to see if the file exists. It works on the file attributes as they exist on disk, not as a program's OPEN statement specifies them.

Instead of ACCESS, you can use the INQUIRE statement with the ACTION specifier to check for similar information. (The ACCESS function always returns 0 for read permission on FAT files, meaning that all files have read permission.)

Date and Time Routines

Various date and time routines are available to determine system time, or convert it to local time, Greenwich Mean Time, arrays of date and time elements, or an ASCII character string.

DATE and TIME are available as either a function or subroutine. Because of the name duplication, if your programs do not include the USE IFPORT statement, each separately compiled program unit can use only one of these versions. For example, if a program calls the subroutine TIME once, it cannot also use TIME as a function.

Standard Fortran 95/90 includes date and time intrinsic subroutines. For more information, see DATE_AND_TIME.

Error Handling Routines

Error handling routines detect and report errors.

IERRNO error codes are analogous to *errno* on Linux* OS and Mac OS* X systems. The IFPORT module provides parameter definitions for many of UNIX's *errno* names, found typically in `errno.h` on UNIX systems.

IERRNO is updated only when an error occurs. For example, if a call to the GETC function results in an error, but two subsequent calls to PUTC succeed, a call to IERRNO returns the error for the GETC call. Examine IERRNO immediately after returning from one of the portability library routines. Other standard Fortran 90 routines might also change the value to an undefined value.

If your application uses multithreading, remember that IERRNO is set on a per-thread basis.

System, Drive, or Directory Control and Inquiry Routines

You can retrieve information about devices, directories, and files with the functions listed below. File names can be long file names or UNC file names. A forward slash in a path name is treated as a backslash. All path names can contain drive specifications.

Standard Fortran 90 provides the INQUIRE statement, which returns detailed file information either by file name or unit number. Use INQUIRE as an equivalent to FSTAT, LSTAT or STAT. LSTAT and STAT return the same information; STAT is the preferred function.

Serial Port Routines (Windows only)

The serial port I/O (SPORT_*xxx*) routines help you perform basic input and output to serial ports. These routines are available only on systems using IA-32 architecture.

Additional Routines

You can also use portability routines for program call and control, keyboards and speakers, file management, arrays, floating-point inquiry and control, IEEE* functionality, and other miscellaneous uses. See the Portability Routines table.

Math Libraries

Intel® Fortran Compiler includes these math libraries:

Library name	Description
`libimf.a` (Linux* OS and Mac OS* X)	Math libraries provided by Intel. This is in addition to `libm.a`, which is the math library provided with gcc* Both of these libraries are linked in by default because certain math functions supported by the GNU* math library are not available in the Intel math library. This linking arrangement allows the GNU users to have all functions available when using ifort, with Intel optimized versions available when supported. `libimf.a` is linked in before `libm.a`. If you link in `libm.a` first, it will change the versions of the math functions that are used.
`libm.lib` (static library) and `libmmd.dll` (the DLL version) (Windows* OS)	Math Libraries provided by Intel.
Intel® Math Kernel Library (Intel® MKL)	Math library of Fortran routines and functions that perform a wide variety of operations on vectors and matrices. The library also includes fast Fourier transform (fft) functions, as well as vector mathematical and vector statistical functions.

Library name	Description
IMSL* Fortran Numerical Library (Windows* OS)	Libraries provided only with certain editions of the Intel® Visual Fortran product. The IMSL* libraries provide a large collection of mathematical and statistical functions accessible from the visual and command line development environments. .

15 Error Handling

Handling Compile Time Errors

Understanding Errors During the Build Process

The Intel® Fortran Compiler identifies syntax errors and violations of language rules in the source program.

Compiler Diagnostic Messages

These messages describe diagnostics that are reported during the processing of the source file. Compiler diagnostic messages usually provide enough information for you to determine the cause of an error and correct it. These messages generally have the following format:

filename(*linenum*:) *severity*: *message*

filename	Indicates the name of the source file currently being processed.
linenum	Indicates the source line where the compiler detects the condition.
severity	Indicates the severity of the diagnostic message: Warning, Error, or Fatal error.
message	Describes the problem.

The following is an example of an error message showing the format and message text:

```
echar.for(7): Severe: Unclosed DO loop or IF block
       DO I=1,5
--------^
```

The pointer (`---^`) indicates the exact place on the source program line where the error was found, in this case where an **END DO** statement was omitted.

To view the passes as they execute on the command line, specify `-watch` (Linux* OS and Mac OS* X) or `/watch` (Windows* OS).

NOTE. You can perform compile-time procedure interface checking between routines with no explicit interfaces present. To do this, generate a module containing the interface for each compiled routine using the `-gen-interfaces` (Linux OS and Mac OS X) or `/gen-interfaces` (Windows OS) option and check implicit interfaces using the `-warn interfaces` (Linux OS and Mac OS X) or `/warn:interfaces` (Windows OS) option.

Controlling Compiler Diagnostic Warning and Error Messages

You can use a number of compiler options to control the diagnostic messages issued by the compiler. For example, the `-WB` (Linux OS and Mac OS X) or `/WB` (Windows OS) compiler option turns compile time bounds errors into warnings. To control compiler diagnostic messages (such as warning messages), use `-warn` (Linux OS and Mac OS X) or `/warn` (Windows OS). The `-warn [keyword]` (Linux OS and Mac OS X) or `/warn:keyword` (Windows OS) option controls warnings issued by the compiler and supports a wide range of values. Some of these are as follows:

[no]alignments -- Determines whether warnings occur for data that is not naturally aligned.

[no]declarations -- Determines whether warnings occur for any undeclared symbols.

[no]errors -- Determines whether warnings are changed to errors.

[no]general -- Determines whether warning messages and informational messages are issued by the compiler.

[no]interfaces -- Determines whether warnings about the interfaces for all called SUBROUTINEs and invoked FUNCTIONs are issued by the compiler.

[no]stderrors -- Determines whether warnings about Fortran standard violations are changed to errors.

[no]truncated_source -- Determines whether warnings occur when source exceeds the maximum column width in fixed-format files.

For more information, see the `-warn` compiler option.

You can also control the display of diagnostic information with variations of the `-diag` (Linux OS and Mac OS X) or `/Qdiag` (Windows OS) compiler option. This compiler option accepts numerous arguments and values, allowing you wide control over displayed diagnostic messages and reports.

Some of the most common variations include the following:

Linux OS and Mac OS X	Windows OS	Description
`-diag-enable` *`list`*	`/Qdiag-enable:`*`list`*	Enables a diagnostic message or a group of messages
`-diag-disable` *`list`*	`/Qdiag-disable:`*`list`*	Disables a diagnostic message or a group of messages
`-diag-warning` *`list`*	`/Qdiag-warning:`*`list`*	Tells the compiler to change diagnostics to warnings
`-diag-erro` *`list`*	`/Qdiag-error:`*`list`*	Tells the compiler to change diagnostics to errors
`-diag-remark` *`list`*	`/Qdiag-remark:`*`list`*	Tells the compiler to change diagnostics to remarks (comments)

The *`list`* items can be specific diagnostic IDs, one of the keywords `warn`, `remark`, or `error`, or a keyword specifying a certain group (`par`, `vec`, `driver`, `cpu-dispatch`, `sv`). For more information, see `-diag`, `/Qdiag`.

Additionally, you can use the following related options:

Linux OS and Mac OS X	Windows OS	Description
`-diag-dump`	`/Qdiag-dump`	Tells the compiler to print all enabled diagnostic messages and stop compilation
`-diag-file[=`*`file`*`]`	`/Qdiag-file[:`*`file`*`]`	Causes the results of diagnostic analysis to be output to a file
`-diag-file-append[=`*`file`*`]`	`/Qdiag-file-append[:`*`file`*`]`	Causes the results of diagnostic analysis to be appended to a file
`-diag-error-limit n`	`/Qdiag-error-limit:n`	Specifies the maximum number of errors allowed before compilation stops.

Linker Diagnostic Errors

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors occur, the linker does not produce an executable file. Linker messages are descriptive, and you do not normally need additional information to determine the specific error.

To view the libraries being passed to the linker on the command line, specify `-watch` or `/watch`.

Error Severity Levels

Comment Messages

These messages indicate valid but inadvisable use of the language being compiled. The compiler displays comments by default. You can suppress comment messages with the `-warn nousage` (Linux OS and Mac OS X) or `/warn:nousage` (Windows OS) option.

Comment messages do not terminate translation or linking, they do not interfere with any output files either. Some examples of the comment messages are:

```
Null CASE construct
The use of a non-integer DO loop variable or expression
Terminating a DO loop with a statement other than CONTINUE or ENDDO
```

Warning Messages

These messages report valid but questionable use of the language being compiled. The compiler displays warnings by default. You can suppress warning messages by using the `-warn` or `/warn` option. Warnings do not stop translation or linking. Warnings do not interfere with any output files. Some representative warning messages are:

```
constant truncated - precision too great
non-blank characters beyond column 72 ignored
Hollerith size exceeds that required by the context
```

Error Messages

These messages report syntactic or semantic misuse of Fortran.

Errors suppress object code for the error containing the error and prevent linking, but they do not stop from parsing to continue to scan for any other errors. Some typical examples of error messages are:

```
line exceeds 132 characters
unbalanced parenthesis
incomplete string
```

Fatal Errors

Fatal messages indicate environmental problems. Fatal error conditions stop translation, assembly, and linking. If a fatal error ends compilation, the compiler displays a termination message on standard error output. Some representative fatal error messages are:

```
Disk is full, no space to write object file
Incorrect number of intrinsic arguments
Too many segments, object format cannot support this many segments
```

Using the Command Line

If you are using the command line, messages are written to the standard error output file.

When using the command line:

- Make sure that the appropriate environment variables have been set by executing the `ifortvars.sh` (Linux OS and Mac OS X) or`IFORTVARS.BAT` (Windows OS) file. For example, this BAT file sets the environment variables for the include and library directory paths. For Windows OS, these environment variables are preset if you use the Fortran Command Prompt window in the Intel® Visual Fortran program folder. For a list of environment variables used by the ifort command during compilation, see Setting Compile-Time Environment Variables.
- Specify the libraries to be linked against using compiler options.
- You can specify libraries (include the path, if needed) as file names on the command line.

Compiler Message Catalog Support

Intel Fortran provides a message catalog that contains various compile-time diagnostic messages.

Use the NLSPATH or %PATH% (Windows OS) environment variable to specify the location of the compiler message catalog. Note that this environment variable, which can be defined to point to multiple paths, is also used to specify the location of the run-time catalog. For more information on run-time library catalogs, see Run-Time Message Display and Format.

Using Source Code Verification

Source Checker Overview

The Intel® Fortran Compiler Professional product provides the following source code analysis features:

- source checker analysis
- parallel lint

Source Checker

The source checker is a compiler feature that provides advanced diagnostics based on detailed analysis of source code. It performs static global analysis to find errors in software that go undetected by the compiler itself. general source code analysis tool that provides an additional diagnostic capability to help you debug your programs. You can use source code analysis options to detect potential errors in your compiled code including:

- incorrect usage of OpenMP* directives
- inconsistent object declarations in different program units
- boundary violations
- uninitialized memory
- memory corruptions
- memory leaks
- incorrect usage of pointers and allocatable arrays
- dead code and redundant executions
- typographical errors or uninitialized variables
- dangerous usage of unchecked input

The source checker can be used to analyze and find issues with source files; these source files need not form a whole program (for instance, you can check a library source). In such cases, due to the lack of full information on usage and modification of global objects, calls to routines, and so forth, analysis will be less exact.

Your code must successfully compile, with no errors, for source code analysis options to take effect.

The intended output from the source checker are useful diagnostics; no executable is generated. Object files and library files generated during source checker analysis cannot be used to generate an executable or a dynamic or static library.

Source checker analysis performs a general overview check of a program for all possible values simultaneously. This is in contrast to run-time checking tools that execute a program with a fixed set of values for input variables; such checking tools cannot easily check all edge effects. By not using a fixed set of input values, the source checker analysis can check for all possible corner cases.

Limitations of Source Checker Analysis

Since the source checker does not perform full interpretation of analyzed programs, it can generate so called false-positive messages. This is a fundamental difference between the compiler and source checker generated errors; in the case of the source checker, you decide whether the generated error is legitimate and needs to be fixed.

Example 1: Incorrect message about division by 0

In this example, possible values for parameter x are {6,3}, for parameter y are {3,0}. If x and y both have the value 3, the expression x-y is equal to 0. The source checker cannot identify that the value 3 for x and y cannot coexist.

```
1 #include <stdio.h>
2
3 double foo(int x, int y) {
4 return 1 / (x - y);
5 }
6
7 int main() {
8 printf("%f\n", foo(6, 3));
9 printf("%f\n", foo(3, 0));
10 return 0;
11 }
```

The source checker issues the following message:

```
f1.c(4): error #12062: possible division by 0
```

Example 2: Incorrect message about uninitialization

This example illustrates how a false positive can appear from conditional statements.

```
1 #include <stdio.h>
2
3 int main(int n) {
4 int j;
5 if (n != 0) {
6 j = n;
7 }
8 if (n != 0) {
 9 printf("%d\n", j);
10 }
11 return 0;
12 }
```

The source checker issues the following message:

```
f1.c(9): error #12144: "j" is possibly uninitialized
```

Parallel Lint

Writing and debugging parallel programs requires specific knowledge and tools. Parallel lint can help in both the development of parallel applications and the parallelizing of existing serial applications. Based on source checker algorithms, parallel lint is a source code analysis capability that performs parallelization analysis of OpenMP* programs. The OpenMP 3.0 standard is supported. OpenMP provides a variety of ways for expressing parallelization. Parallel lint can diagnose problems with OpenMP directives and clauses, including:

- nested parallel regions including dynamic extent of parallel regions
- private/shared/reduction variables in parallel regions
- threadprivate variables
- expressions used in OpenMP clauses

In certain cases, an OpenMP program can meet all requirements of the specification but still have serious semantic issues. Parallel lint can help diagnose:

- some types of deadlocks

- data races or potential data dependency
- side effects without proper synchronization

Parallel lint also performs interprocedural analysis and can find issues with parallel directives located in different procedures or files.

Using the Source Code Analysis Options

Source code analysis options include the following:

Option	Result
`-diag-enable sc{[1|2|3]}` (Linux* OS and Mac OS* X) `/Qdiag-enable:sc{[1|2|3]}` (Windows* OS)	Enables source checker analysis. The number specifies the several level of the diagnostics (1=all critical errors, 2=all errors, and 3=all errors and warnings)
`-diag-disable sc` `/Qdiag-disable:sc`	Disables source checker analysis
`-diag-enable sc-parallel{[1|2|3]}` `/Qdiag-enable:sc-parallel{[1|2|3]}`	Enables parallel lint analysis. The number specifies the several level of the diagnostics (1=all critical errors, 2=all errors, and 3=all errors and warnings)
`-diag-disable sc-parallel` `/Qdiag-disable:sc-parallel`	Disables parallel lint analysis
`-diag-enable sc-include` `/Qdiag-enable:sc-include`	Analyzes include files as well as source files.
`-diag-disable warn` `/Qdiag-disable:warn`	Suppresses all warnings, cautions and comments (issues errors only), including those specific to source code analysis

Option	Result
`-diag-disable num-list` `/Qdiag-disable:num-list`	Suppresses messages by number list, where *num-list* is either a single message or a list of message numbers separated by commas and enclosed in parentheses
`-diag-file [file]` `/Qdiag-file[:file]`	Directs diagnostic results to file with `.diag` as the default extension. You need to enable source code analysis diagnostics before you can send them to a file. If a file name is not specified, the diagnostics are sent to *name-of-the-first-source-file*`.diag`.
`-diag-file-append[=file]` `/Qdiag-file-append[:file]`	Appends diagnostic results to file with `.diag` as the default extension. You need to enable source code analysis diagnostics before you can send the results to a file. If you do not specify a path, the current working directory will be searched. If the named file is not found, a new file with that name will be created. If a file name is not specified, the diagnostics are sent to *name-of-the-first-source-file*`.diag`.

The following notes apply:

- Parallel lint diagnostics are a subset of source checker diagnostics. If both the source checker and parallel lint options are specified, only source checker options are used.
- When using parallel lint, be sure to specify the OpenMP compiler option. Add the `-openmp` (Linux OS and Mac OS X) or `/Qopenmp` (Windows OS) option to the command line.
- Specify the `-diag-enable sc-include` (Linux OS and Mac OS X) or `/Qdiag-enable:sc-include` (Windows OS) option to analyze both source files and include files.

Using Source Code Analysis Options with Other Compiler Options

If the `-c` (Linux OS and Mac OS X) or `/c` (Windows OS) compiler option is used on the command line along with a command line option to enable source code analysis, an object file is created; source code analysis diagnostics are not produced. This object file may be used in

a further invocation of source code analysis. To receive complete source code diagnostics, specify source code analysis options for both compilation and linking phases. This feature is useful when a program consists of files written in different languages (C/C++ and Fortran).

```
icc -c -diag-enable sc2 file1.c

ifort -c -diag-enable sc2 file2.f90

icc -diag-enable sc2 file1.obj file2.obj
```

To analyze OpenMP directives, add the `-openmp` (Linux OS and Mac OS X) or `/Qopenmp` (Windows OS) option to the command line.

Using Source Code Analysis within the IDE

When source code analysis support is enabled within the IDE, the customary final build target (e.g. an executable image) is not created. Therefore, you should create a separate "Source Code Analysis" configuration.

In the Eclipse* IDE, do the following:

1. Open the property pages for the project and select **C/C++ Build** .
2. Click the **Manage...** button.
3. In the **Manage** dialog box, click the **New...** button to open the **Create configuration** dialog box.
4. Supply a name for the new configuration in the **Name** box; for example, **Source Code Analysis**.
5. Supply a **Description** for the configuration if you want (optional).
6. You can choose to **Copy settings from** a **Default configuration** or an **Existing configuration** by clicking the appropriate radio button and then selecting a configuration from the corresponding drop down menu.
7. Click **OK** to close the **Create configuration** dialog box.
8. Click **OK** to close the **Manage** dialog box (with your new configuration name selected).

The property pages will now display the settings for your new configuration; this becomes the active build configuration. Navigate to the Intel compiler's **Compilation Diagnostics** properties. Use the **Level of Source Code Parallelization Analysis**, **Level of Source Code Analysis**, and **Analyze Include Files** properties to control source code analysis.

Source Checker Capabilities

Source checker analysis capabilities include the following:

- Interprocedural Analysis for detecting inconsistent objects declarations in different program units and for propagation of data (for example, pointer aliases and argument constant values) using procedure calls
- Local Program Analysis for analyzing each program unit separately and checking for various kinds of problems that will errors or warnings.
- C/C++ specific Analysis for analyzing C/C++ source code and checking for C/C++ specific error and warning conditions. Source code analysis also detects improper code style and flaws in object-oriented design solutions.
- Fortran-specific Analysis for analyzing Fortran source code and checking for Fortran-specific error and warning conditions.
- OpenMP* Analysis for checking for OpenMP API restrictions.

Interprocedural Analysis

Source code analysis detects inconsistent object declarations in different program units, for example:

- Different external objects with the same name.
- Inconsistent declarations of a COMMON block (Fortran-specific).
- Mismatched number of arguments.
- Mismatched type, rank, shape, or/and size of an argument.
- Inconsistent declaration of a procedure.

The following examples illustrate interprocedural analysis.

Example 1: Wrong number of arguments

File f1.c contains this function declaration:

```
1  void Do_hello() {
2      printf("helo everybody!\n");
}
```

File f2.c contains a call to the routine:

```
1 extern void Do_hello(int);

2

3 void foo() {

4 Do_hello ( 1 ) ;

 5 }
```

The source checker issues the following message:

```
f2.c(4): error #12020: number of actual arguments (1) in call of "Do_hello"
doesn't match the number of formal arguments (0); "Do_hello" is defined at
(file:f1.c line:1)
```

The other goal of interprocedural analysis is to propagate information about program objects across the program through procedure calls. The following information is propagated :

- Ranges of constant values
- Pointer attributes
- Information about usage and modification of objects

Example 2: Out of Boundaries

This example demonstrates the propagation of pointer A to Arr and the constant value 5 to x:

```
1 void foo( int* Arr, int x) {

2 for(; 0<=x; x--) {

3 Arr[x] = x;

4 }

5 }

6

7 void bar() {

8 int A[5];

9 foo(A,5);

10 }
```

The source checker issues the following message:

```
f1.c(3): error #12255: Buffer overflow: index is possibly outside the bounds
for array "A" which is passed as actual argument 1 to "foo" at (file:f2.c
line:9); array "A" of size (0:4) can be indexed by value 5
```

Local Program Analysis

The source checker uses local analysis of each program unit to check for various kinds of errors, warnings, and/or debatable points in a program. Examples of these errors are:

- Incorrect use or modification of an object
- Problems with memory (for example, leaks, corruptions, uninitialized memory)
- Incorrect use with pointers
- Boundaries violations
- Wrong value of an argument in an intrinsic call
- Dead code and redundant executions

The following examples illustrate local program analysis.

Example 1: Object is smaller than required size

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(void){
5 char string[10];
6
7 strcpy(string, "Hello world from");
8 printf("%s\n",string);
9
10 return 0;
11 }
```

The following message is issued :

```
f1.c(7): error #12224: Buffer overflow: size of object "string" (10 bytes)
is less than required size (17 bytes)
```

Example 2: Memory Leak

File `f1.c` contains the following:

```
1 #include <stdio.h>
2 #include <malloc.h>
3
4 int main(void) {
5 float **ptr;
6
7 ptr = (float **)malloc(8);
8 if (ptr == NULL) exit(1);
9 *ptr = (float*)malloc(sizeof(float));
10 if (*ptr == NULL) exit(1);
11 **ptr = 3.14;
12 printf("%f\n",**ptr);
13 free(ptr);
14 return 0;
15 }
```

The source checker issues the following message:

```
f1.c(14): error #12121: memory leak: dynamic memory allocated at (file:f1.c
line:9) is not freed at this point
```

Fortran-specific Analysis

The source checker is able to detect issues with the following:

- Mismatched type, rank, shape, or/and size of an argument
- Incorrect usage of ALLOCATABLE arrays
- Inconsistency in COMMON blocks

The following example illustrates Fortran-specific analysis.

Example 1: Undefined function result

File f1.f contains the following lines:

```
1 subroutine foo(m)
2 integer, dimension(2,3) :: m
```

```
3 do i=1,3
4 print *,m(:,i)
5 end do
6 end
7 integer, dimension(3,2) :: n
8 do i=1,2
9 n(:,i) = i
10 end do
11 call foo(n)
12 ! shapes of argument #1 and dummy argument are different.
13 do i=1,2
14 print *,n(:,i)
15 end do
16 end
```

Source code analysis issues the following message:

```
f1.f(11): error #12028: shape of actual argument 1 in call of "FOO" doesn't
match the shape of formal argument "M"; "FOO" is defined
```

C/C++ specific Analysis

Source code analysis examines C/C++ source code and checks for C++ specific errors. It also points out places of improper code style and flaws in object-oriented design solutions.

The source checker detects issues with the following:

- Memory management (leaks, mixing C and C++ memory management routines, smart pointer usage)
- C++ exception handling (uncaught exception, exception from destructor/operator delete)
- Misuse of operator new/operator delete
- Misuse of virtual functions

Example 1: Call of virtual function from constructor

```
1 #include "stdio.h"
2
3 class A {
4 public:
5 A() { destroy(); }
6 void destroy() { clear0();}
7 virtual void clear()=0;
8 void clear0() { clear(); };
9 };
10
11 class B : public A {
12 public:
13 B(){ }
14 virtual void clear(){ printf("overloaded clear"); }
15 virtual ~B() { }
16 };
17
18 int main() {
19 B b;
20 return 0;
21 }
```

The source checker issues the following message:

```
f1.cpp(8): warning #12327: pure virtual function "clear" is called from
constructor (file:f1.cpp line:5)
```

OpenMP* Analysis

The compiler detects some restrictions noted in the OpenMP* API. With OpenMP analysis, additional checks for misuse of the OpenMP API are performed.

Example 1: Incorrect usage of OpenMP directives

File f1.c contains the following lines:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 void fff(int ii) {
5 #pragma omp barrier
6 printf("Val = %d \n", ii);
7 }
8
9 int main(void) {
10 int i=3;
11 omp_set_num_threads(3);
12 #pragma omp parallel
13 #pragma omp master
14 fff(i);
15 return 0;
16 }
```

Source code analysis issues the following message:

```
f1.c(5): error #12200: BARRIER directive is not allowed in the dynamic extent
of MASTER directive (file:f1.c line:13)
```

Example 2: Incorrect data dependency

To enable data dependency analysis for a parallel program, enable diagnostics at level 3 .

```
1 int main(void) {
2 int i,sum = 0;
3 int a[1000];
4
5 #pragma omp parallel for reduction(+:sum)
6 for (i=1; i<999; i++) {
7 a[i] = i;
8 sum = sum + a[i + 1];
9 }
10 }
```

Source code analysis issues the following message:

```
f1.c(8): warning #12247: anti data dependence from (file:f1.c line:8) to
(file:f1.c line:7), due to "a" may lead to incorrect program execution in
parallel mode
```

Example 3: Incorrect synchronization

File f1.c contains the following lines:

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int a[1000];
5 int sum = 0;
6
7 void moo() {
8 int i;
9
10 #pragma omp task
11 for (i=0; i<1000; i++) {
12 a[i] = i;
13 sum = sum + a[i];
14 }

15 }

16

17 void foo() {

18 printf("%d\n",sum);

19 }

20

21 int main(void) {

22 int i;
```

```
23

24 #pragma omp parallel shared(sum)

25 #pragma omp single

26 {

27 moo();

28 foo();

29 }

30 return 0;

31 }
```

Source code analysis issues the following message:

```
f1.c(18): error #12365: variable "sum" is defined at (file:f1.c line:13) in
TASK region (file:f1.c line:10) and is used before synchronization
```

Handling Run-Time Errors

Understanding Run-Time Errors

During execution, your program may encounter errors or exception conditions. These conditions can result from any of the following:

- Errors that occur during I/O operations
- Invalid input data
- Argument errors in calls to the mathematical library
- Arithmetic errors
- Other system-detected errors

The Intel® Fortran run-time system (Run-Time Library or RTL) generates appropriate messages and takes action to recover from errors whenever possible.

For a description of each Intel Fortran run-time error message, see List of Run-Time Error Messages.

There are a few tools and aids that are helpful when an application fails and you need to diagnose the error. Compiler-generated machine code listings and linker-generated map files can help you understand the effects of compiler optimizations and to see how your application is laid out in memory. They may help you interpret the information provided in a stack trace at the time of the error. See Generating Listing and Map Files.

Forcing a Core Dump for Severe Errors

You can force a core dump for severe errors that do not usually cause a `core` file to be created. Before running the program, set the `DECFORT_DUMP_FLAG` environment variable to any of the common TRUE values (Y, y, Yes, yEs, True, and so forth) to cause severe errors to create a `core` file. For instance, the following C shell command sets the `DECFORT_DUMP_FLAG` environment variable:

```
setenv decfort_dump_flag y
```

The `core` file is written to the current directory and can be examined using a debugger.

NOTE. If you requested a core file to be created on severe errors and you don't get one when expected, the problem might be that your process limit for the allowable size of a core file is set too low (or to zero). See the man page for your shell for information on setting process limits. For example, the C shell command limit (with no arguments) will report your current settings, and `limit coredumpsize unlimited` will raise the allowable limit to your current system maximum.

See Also

- Handling Run-Time Errors
- Run-Time Default Error Processing
- Run-Time Message Display and Format
- Values Returned at Program Termination
- Methods of Handling Errors
- Using the END, EOR, and ERR Branch Specifiers
- Using the IOSTAT Specifier and Fortran Exit Codes
- Locating Run-Time Errors
- List of Run-Time Error Messages
- Signal Handling
- Overriding the Default Run-Time Library Exception Handler

- Using Traceback Information

Run-Time Default Error Processing

The Intel® Fortran run-time system processes a number of errors that can occur during program execution. A default action is defined for each error recognized by the Intel Fortran run-time system. The default actions described throughout this section occur unless overridden by explicit error-processing methods.

The way in which the Intel Fortran run-time system actually processes errors depends upon the following factors:

- The severity of the error. For instance, the program usually continues executing when an error message with a severity level of warning or info (informational) is detected.
- For certain errors associated with I/O statements, whether or not an I/O error-handling specifier was specified.
- For certain errors, whether or not the default action of an associated signal was changed.
- For certain errors related to arithmetic operations (including floating-point exceptions), compilation options can determine whether the error is reported and the severity of the reported error.

How arithmetic exception conditions are reported and handled depends on the cause of the exception and how the program was compiled. Unless the program was compiled to handle exceptions, the exception might not be reported until *after* the instruction that caused the exception condition.

See Also

- Handling Run-Time Errors
- Run-Time Message Display and Format
- Values Returned at Program Termination
- Locating Run-Time Errors
- Using Traceback Information
- Data Representation

Run-Time Message Display and Format

Fortran run-time messages have the following format:

forrtl: *severity* (*number*): *message-text*

where:

- *forrtl*

Identifies the source as the Intel Fortran run-time system (Run-Time Library or RTL).

- *severity*

  The severity levels are: *severe.info* , or *warning*,*error* ,

- *number*

  This is the message number; also the IOSTAT value for I/O statements.

- *message-text*

  Explains the event that caused the message.

The following table explains the severity levels of run-time messages, in the order of greatest to least severity. The severity of the run-time error message determines whether program execution continues:

Severity	Description
severe	Must be corrected. The program's execution is terminated when the error is encountered unless the program's I/O statements use the END, EOR, or ERR branch specifiers to transfer control, perhaps to a routine that uses the IOSTAT specifier. (See Using the END, EOR, and ERR Branch Specifiers and Using the IOSTAT Specifier and Fortran Exit Codes and Methods of Handling Errors.) For severe errors, stack trace information is produced by default, unless the environment variable FOR_DISABLE_STACK_TRACE is set. If the command line option -traceback (Linux* OS and Mac OS* X) or /traceback (Windows* OS) is specified, the stack trace information contains program counters set to symbolic information. Otherwise, the information contains merely hexadecimal program counter information. In some cases stack trace information is also produced by the compiled code at run-time to provide details about the creation of array temporaries. If FOR_DISABLE_STACK_TRACE is set, no stack trace information is produced.
error	Should be corrected. The program might continue execution, but the output from this execution might be incorrect. For errors of severity type error, stack trace information is produced by default, unless the environment variable FOR_DISABLE_STACK_TRACE is set. If the command line option -traceback (Linux OS and Mac OS X) or /traceback (Windows) is specified, the stack trace information contains program counters set to symbolic information. Otherwise, the information contains merely hexadecimal program counter information. In some cases stack trace information is also produced by the compiled code at run-time to provide details about the creation of array temporaries.

Severity	Description
	If FOR_DISABLE_STACK_TRACE is set, no stack trace information is produced.
warning	Should be investigated. The program continues execution, but output from this execution might be incorrect.
info	For informational purposes only; the program continues.

For a description of each Intel Fortran run-time error message, see Run-Time Default Error Processing and related topics.

The following example applies to Linux OS and Mac OS X:

In some cases, stack trace information is produced by the compiled code at run time to provide details about the creation of array temporaries.

(If FOR_DISABLE_STACK_TRACE is set, no stack trace information is produced.)

The following program generates an error at line 12:

```
program ovf
real*4 x(5),y(5)
integer*4 i

x(1) = -1e32
x(2) = 1e38
x(3) = 1e38
x(4) = 1e38
x(5) = -36.0

do i=1,5
y(i) = 100.0*(x(i))
print *, 'x = ', x(i), ' x*100.0 = ',y(i)
end do
end
```

The following command line produces stack trace information for the program executable.

```
> ifort -O0 -fpe0 -traceback ovf.f90 -o ovf.exe
> ovf.exe

x =  -1.0000000E+32  x*100.0
 =  -1.0000000E+34
forrtl: error (72): floating overflow
Image        PC         Routine       Line      Source
ovf.exe      08049E4A   MAIN__            14    ovf.f90
ovf.exe      08049F08   Unknown       Unknown   Unknown
ovf.exe      400B3507   Unknown       Unknown   Unknown
ovf.exe      08049C51   Unknown       Unknown   Unknown
Abort
```

The following suppresses stack trace information because the FOR_DISABLE_STACK_TRACE environment variable is set.

```
> setenv FOR_DISABLE_STACK_TRACE true
> ovf.exe

x =  -1.0000000E+32  x*100.0 =  -1.0000000E+34
forrtl: error (72): floating overflow
Abort
```

Run-Time Library Message Catalog File Location

The libifcore, libirc, and libm run-time libraries ship message catalogs. When a message by one of these libraries is to be displayed, the library searches for its message catalog in a directory specified by either the `NLSPATH` (Linux OS and Mac OS X), or `%PATH%` (Windows OS) environment variable. If the message catalog cannot be found, the message is displayed in English.

The names of the three message catalogs are as follows:

libifcore message catalogs and related text message files	Linux OS and Mac OS X	ifcore_msg.cat ifcore_msg.msg
	Windows OS	ifcore_msg.dll ifcore_msg.mc
libirc message catalogs and related text message files	Linux OS and Mac OS X	irc_msg.cat irc_msg.msg
	Windows OS	irc_msg.dll irc_msg.mc
libm message catalogs and related text message files	Linux OS and Mac OS X	libm.cat libm.msg
	Windows OS	libmUI.dll libmUI.mc

Values Returned at Program Termination

An Intel Fortran program can terminate in a number of ways. On Linux OS and Mac OS X, values are returned to the shell.

- The program runs to normal completion. A value of zero is returned.
- The program stops with a `STOP` statement. If an integer stop-code is specified, a status equal to the code is returned; if no stop-code is specified, a status of zero is returned.
- The program stops because of a signal that is caught but does not allow the program to continue. A value of 1 is returned.
- The program stops because of a severe run-time error. The error number for that run-time error is returned. See Understanding Run-Time Errors and related topics.
- The program stops with a **`CALL EXIT`** statement. The value passed to **`EXIT`** is returned.
- The program stops with a `CALL ABORT` statement. A value of 134 is returned.

Methods of Handling Errors

Whenever possible, the Intel® Fortran RTL does certain error handling, such as generating appropriate messages and taking necessary action to recover from errors. You can explicitly supplement or override default actions by using the following methods:

- To transfer control to error-handling code within the program, use the END, EOR, and ERR branch specifiers in I/O statements. See Using the END, EOR, and ERR Branch Specifiers.
- To identify Fortran-specific I/O errors based on the value of Intel Fortran RTL error codes, use the I/O status specifier (IOSTAT) in I/O statements (or call the `ERRSNS` subroutine). See Using the IOSTAT Specifier and Fortran Exit Codes.
- Obtain system-level error codes by using the appropriate library routines.
- For certain error conditions, use the signal handling facility to change the default action to be taken.

Using the END, EOR, and ERR Branch Specifiers

When a severe error occurs during Intel® Fortran program execution, the default action is to display an error message and terminate the program. To override this default action, there are three branch specifiers you can use in I/O statements to transfer control to a specified point in the program:

- The END branch specifier handles an end-of-file condition.
- The EOR branch specifier handles an end-of-record condition for nonadvancing reads.
- The ERR branch specifier handles all error conditions.

If you use the END, EOR, or ERR branch specifiers, no error message is displayed and execution continues at the designated statement, usually an error-handling routine.

You might encounter an unexpected error that the error-handling routine cannot handle. In this case, do one of the following:

- Modify the error-handling routine to display the error message number.
- Remove the END, EOR, or ERR branch specifiers from the I/O statement that causes the error.

After you modify the source code, compile, link, and run the program to display the error message. For example:

```
READ (8,50,ERR=400)
```

If any severe error occurs during execution of this statement, the Intel Visual Fortran RTL transfers control to the statement at label 400. Similarly, you can use the END specifier to handle an end-of-file condition that might otherwise be treated as an error. For example:

```
READ (12,70,END=550)
```

When using nonadvancing I/O, use the EOR specifier to handle the end-of-record condition. For example:

```
150 FORMAT (F10.2, F10.2, I6)
    READ (UNIT=20, FMT=150, SIZE=X, ADVANCE='NO', EOR=700) A, F, I
```

You can also use ERR as a specifier in an OPEN, CLOSE, or INQUIRE statement. For example:

```
OPEN (UNIT=10, FILE='FILNAM', STATUS='OLD', ERR=999)
```

If an error is detected during execution of this `OPEN` statement, control transfers to the statement at label 999.

Using the IOSTAT Specifier and Fortran Exit Codes

You can use the IOSTAT specifier to continue program execution after an I/O error and to return information about I/O operations. Certain errors are *not* returned in IOSTAT.

The IOSTAT specifier can supplement or replace the END, EOR, and ERR branch transfers.

Execution of an I/O statement containing the IOSTAT specifier suppresses the display of an error message and defines the specified integer variable, array element, or scalar field reference as one of the following, which is returned as an exit code if the program terminates:

- A value of -2 if an end-of-record condition occurs with nonadvancing reads.
- A value of -1 if an end-of-file condition occurs.
- A value of 0 for normal completion (not an error condition, end-of-file, or end-of-record condition).
- A positive integer value if an error condition occurs. (This value is one of the Fortran-specific IOSTAT numbers listed in the run-time error message. See List of Run-Time Error Messages, which lists the messages.)

Following the execution of the I/O statement and assignment of an IOSTAT value, control transfers to the END, EOR, or ERR statement label, if any. If there is no control transfer, normal execution continues.

You can include the `for_iosdef.for` file in your program to obtain symbolic definitions for the values of IOSTAT.

The following example uses the IOSTAT specifier and the `for_iosdef.for` file to handle an `OPEN` statement error (in the FILE specifier).

Error Handling OPEN Statement File Name

```
     CHARACTER(LEN=40) :: FILNM
    INCLUDE 'for_iosdef.for'
    DO I=1,4      FILNM = ''
      WRITE (6,*)  'Type file name '
      READ (5,*) FILNM
      OPEN (UNIT=1, FILE=FILNM, STATUS='OLD', IOSTAT=IERR, ERR=100)
      WRITE (6,*) 'Opening file: ', FILNM
!       (process the input file)
      CLOSE (UNIT=1)
      STOP
  100  IF (IERR .EQ. FOR$IOS_FILNOTFOU) THEN
        WRITE (6,*) 'File: ', FILNM, ' does not exist '
      ELSE IF (IERR .EQ. FOR$IOS_FILNAMSPE) THEN
         WRITE (6,*) 'File: ', FILNM, ' was bad, enter new file name'
      ELSE
        PRINT *, 'Unrecoverable error, code =', IERR
        STOP
      END IF
    END DO
    WRITE (6,*) 'File not found. Locate correct file with Explorer and run again'
  END PROGRAM
```

Another way to obtain information about an error is the ERRSNS subroutine, which allows you to obtain the last I/O system error code associated with an Intel Fortran RTL error (see the *Intel® Fortran Language Reference*).

Locating Run-Time Errors

This topic provides some guidelines for locating the cause of exceptions and run-time errors. Intel Fortran run-time error messages do not usually indicate the exact source location causing the error. The following compiler options are related to handling errors and exceptions:

- The `-check [keyword]` (Linux OS Mac OS X) or `/check[:keyword]` (Windows OS) option generates extra code to catch certain conditions at run time. For example, if you specify the keyword of `bounds`, the debugger will catch and stop at array or character string bounds errors. You can specify the keyword of `bounds` to generate code to perform compile-time and run-time checks on array subscript and character substring expressions. An error is

reported if the expression is outside the dimension of the array or the length of the string. The keyword of `uninit` generates code for dynamic checks of uninitialized variables. If a variable is read before written, a run-time error routine will be called. The `noformat` and `nooutput_conversion` keywords reduce the severity level of the associated run-time error to allow program continuation. The `pointers` keyword generates code to test for disassociated pointers and unallocatable arrays.

The following `-check pointers` (Linux OS and Mac OS X) or `/check:pointers` (Windows OS) examples result in various output messages.

Example 1: Allocatable variable not allocated

```
        real, allocatable:: a(:)
!       allocate(a(4))  ! if a is unallocated, the next statement gets an error with
"check pointers"
        a=17
        print  *,a
        end
 Output 1:
forrtl: severe (408): fort: (8): Attempt to fetch from allocatable variable A when it
is not allocated
```

Example 2: Pointer not associated

```
        real, pointer:: a(:)
        allocate(a(5))
        a=17
        print  *,a
        deallocate(a)   ! once a is deallocated, the next statement gets an error with
"check pointers"
        a=20
        print  *,a
        end

Output 2:
   17.00000       17.00000       17.00000       17.00000       17.00000
forrtl: severe (408): fort: (7): Attempt to use pointer A when it is not associated with
 a target
```

Example 3: Cray pointer with zero value

```
        pointer(p,a)
        real, target:: b
!       p=loc(b)        ! if integer pointer p has no address assigned to it,
!                       ! the next statement gets an error with "check pointers"
        b=17.
        print  *,a
        end

Output 3:
forrtl: severe (408): fort: (9): Attempt to use pointee A when its corresponding integer
 pointer P has the value zero
```

- The `-ftrapuv` (Linux OS and Mac OS X) or `/Qtrapuv` (Windows OS) option is useful in detecting uninitialized variables. It sets uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables, which are not properly initialized by the application, are likely to cause run-time errors that can help you detect coding errors.
- The `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS) option generates extra information in the object file to provide source file traceback information when a severe error occurs at run time. This simplifies the task of locating the cause of severe run-time errors. Without `traceback`, you could try to locate the cause of the error using a map file and the hexadecimal addresses of the stack displayed when a severe error occurs.Certain traceback-related information accompanies severe run-time errors, as described in Using Traceback Information.
- The `-fpe` (Linux OS and Mac OS X) or `/fpe` (Windows OS) option controls the handling of floating-point arithmetic exceptions (IEEE arithmetic) at run time. If you specify the `-fpe3` (Linux OS and Mac OS X) or `/fpe:3` (Windows OS) compiler option, all floating-point exceptions are disabled, allowing IEEE exceptional values and program continuation. In contrast, specifying `-fpe0` or `/fpe:0` stops execution when an exceptional value (such as a NaN) is generated or when attempting to use a denormalized number, which usually allows you to localize the cause of the error.
- The `-warn` and `-nowarn` (Linux OS and Mac OS X) or `/warn` and `/nowarn` (Windows OS) options control compile-time warning messages, which, in some circumstances, can help determine the cause of a run-time error.
- On Linux OS and Mac OS X, the `-fexceptions` option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines.
- On Windows OS, the Compilation Diagnostics Options in the IDE control compile-time diagnostic messages, which, in some circumstances can help determine the cause of a run-time error.

See Also

- Handling Run-Time Errors
- Understanding Run-Time Errors

List of Run-Time Error Messages

This section lists the errors processed by the Intel Fortran run-time library (RTL). For each error, the table provides the error number, the severity code, error message text, condition symbol name, and a detailed description of the error.

To define the condition symbol values (PARAMETER statements) in your program, include the following file:

`for_iosdef.for`

As described in the table, the severity of the message determines which of the following occurs:

- with `info` and `warning`, program execution continues
- with `error`, the results may be incorrect
- with `severe`, program execution stops (unless a recovery method is specified)

In the last case, to prevent program termination, you must include either an appropriate I/O error-handling specifier and recompile or, for certain errors, change the default action of a signal before you run the program again.

In the following table, the first column lists error numbers returned to IOSTAT variables when an I/O error is detected.

The first line of the second column provides the message as it is displayed (following *`forrtl:`*), including the severity level, message number, and the message text. The following lines of the second column contain the status condition symbol (such as FOR$IOS_INCRECTYP) and an explanation of the message.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
1[1]	**severe (1): Not a Fortran-specific error** FOR$IOS_NOTFORSPE. An error in the user program or in the RTL was not an Intel Fortran-specific error and was not reportable through any other Intel Fortran run-time messages.
8	**severe (8): Internal consistency check failure** FOR$IOS_BUG_CHECK. Internal error. Please check that the program is correct. Recompile if an error existed in the program. If this error persists, submit a problem report.
9	**severe (9): Permission to access file denied** FOR$IOS_PERACCFIL. Check the permissions of the specified file and whether the network device is mapped and available. Make sure the correct file and device was being accessed. Change the protection, specific file, or process used before rerunning the program.
10	**severe (10): Cannot overwrite existing file**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_CANOVEEXI. Specified file xxx already exists when OPEN statement specified STATUS='NEW' (create new file) using I/O unit x. Make sure correct file name, directory path, unit, and so forth were specified in the source program. Decide whether to: • Rename or remove the existing file before rerunning the program. • Modify the source file to specify different file specification, I/O unit, or OPEN statement STATUS.
11[1]	**info (11): Unit not connected** FOR$IOS_UNINOTCON. The specified unit was not open at the time of the attempted I/O operation. Check if correct unit number was specified. If appropriate, use an OPEN statement to explicitly open the file (connect the file to the unit number).
17	**severe (17): Syntax error in NAMELIST input** FOR$IOS_SYNERRNAM. The syntax of input to a namelist-directed READ statement was incorrect.
18	**severe (18): Too many values for NAMELIST variable** FOR$IOS_TOOMANVAL. An attempt was made to assign too many values to a variable during a namelist READ statement.
19	**severe (19): Invalid reference to variable in NAMELIST input** FOR$IOS_INVREFVAR. One of the following conditions occurred: • The variable was not a member of the namelist group. • An attempt was made to subscript a scalar variable. • A subscript of the array variable was out-of-bounds. • An array variable was specified with too many or too few subscripts for the variable. • An attempt was made to specify a substring of a noncharacter variable or array name. • A substring specifier of the character variable was out-of-bounds. • A subscript or substring specifier of the variable was not an integer constant. • An attempt was made to specify a substring by using an unsubscripted array variable.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
20	**severe (20): REWIND error** FOR$IOS_REWERR. One of the following conditions occurred: • The file was not a sequential file. • The file was not opened for sequential or append access. • The Intel Fortran RTL I/O system detected an error condition during execution of a REWIND statement.
21	**severe (21): Duplicate file specifications** FOR$IOS_DUPFILSPE. Multiple attempts were made to specify file attributes without an intervening close operation. A DEFINE FILE statement was followed by another DEFINE FILE statement or an OPEN statement.
22	**severe (22): Input record too long** FOR$IOS_INPRECTOO. A record was read that exceeded the explicit or default record length specified when the file was opened. To read the file, use an OPEN statement with a RECL= value (record length) of the appropriate size.
23	**severe (23): BACKSPACE error** FOR$IOS_BACERR. The Intel Fortran RTL I/O system detected an error condition during execution of a BACKSPACE statement.
24[1]	**severe (24): End-of-file during read** FOR$IOS_ENDDURREA. One of the following conditions occurred: • An Intel Fortran RTL I/O system end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An end-of-file record written by the ENDFILE statement was encountered during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. • An attempt was made to read past the end of an internal file character string or array during execution of a READ statement that did not contain an END, ERR, or IOSTAT specification. This error is returned by END and ERRSNS.
25	**severe (25): Record number outside range**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_RECNUMOUT. A direct access READ, WRITE, or FIND statement specified a record number outside the range specified when the file was opened.
26	**severe (26): OPEN or DEFINE FILE required** FOR$IOS_OPEDEFREQ. A direct access READ, WRITE, or FIND statement was attempted for a file when no prior DEFINE FILE or OPEN statement with ACCESS='DIRECT' was performed for that file.
27	**severe (27): Too many records in I/O statement** FOR$IOS_TOOMANREC. An attempt was made to do one of the following: • Read or write more than one record with an ENCODE or DECODE statement. • Write more records than existed.
28	**severe (28): CLOSE error** FOR$IOS_CLOERR. An error condition was detected by the Intel Fortran RTL I/O system during execution of a CLOSE statement.
29	**severe (29): File not found** FOR$IOS_FILNOTFOU. A file with the specified name could not be found during an open operation.
30	**severe (30): Open failure** FOR$IOS_OPEFAI. An error was detected by the Intel Fortran RTL I/O system while attempting to open a file in an OPEN, INQUIRE, or other I/O statement. This message is issued when the error condition is not one of the more common conditions for which specific error messages are provided. It can occur when an OPEN operation was attempted for one of the following: • Segmented file that was not on a disk or a raw magnetic tape • Standard I/O file that had been closed
31	**severe (31): Mixed file access modes** FOR$IOS_MIXFILACC. An attempt was made to use any of the following combinations: • Formatted and unformatted operations on the same unit • An invalid combination of access modes on a unit, such as direct and sequential

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• An Intel Fortran RTL I/O statement on a logical unit that was opened by a program coded in another language
32	**severe (32): Invalid logical unit number** FOR$IOS_INVLOGUNI. A logical unit number greater than 2,147,483,647 or less than zero was used in an I/O statement.
33	**severe (33): ENDFILE error** FOR$IOS_ENDFILERR. One of the following conditions occurred: • The file was not a sequential organization file with variable-length records. • The file was not opened for sequential, append, or direct access. • An unformatted file did not contain segmented records. • The Intel Fortran RTL I/O system detected an error during execution of an ENDFILE statement.
34	**severe (34): Unit already open** FOR$IOS_UNIALROPE. A DEFINE FILE statement specified a logical unit that was already opened.
35	**severe (35): Segmented record format error** FOR$IOS_SEGRECFOR. An invalid segmented record control data word was detected in an unformatted sequential file. The file was probably either created with RECORDTYPE='FIXED' or 'VARIABLE' in effect, or was created by a program written in a language other than Fortran or Fortran 90.
36	**severe (36): Attempt to access non-existent record** FOR$IOS_ATTACCNON. A direct-access `READ` or `FIND` statement attempted to access beyond the end of a relative file (or a sequential file on disk with fixed-length records) or access a record that was previously deleted from a relative file.
37	**severe (37): Inconsistent record length** FOR$IOS_INCRECLEN. An attempt was made to open a direct access file without specifying a record length.
38	**severe (38): Error during write**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_ERRDURWRI. The Intel Fortran RTL I/O system detected an error condition during execution of a WRITE statement.
39	**severe (39): Error during read** FOR$IOS_ERRDURREA. The Intel Fortran RTL I/O system detected an error condition during execution of a READ statement.
40	**severe (40): Recursive I/O operation** FOR$IOS_RECIO_OPE. While processing an I/O statement for a logical unit, another I/O operation on the same logical unit was attempted, such as a function subprogram that performs I/O to the same logical unit that was referenced in an expression in an I/O list or variable format expression.
41	**severe (41): Insufficient virtual memory** FOR$IOS_INSVIRMEM. The Intel Fortran RTL attempted to exceed its available virtual memory while dynamically allocating space. To overcome this problem, investigate increasing the data limit. Before you try to run this program again, wait until the new system resources take effect. Note: This error can be returned by STAT in an ALLOCATE or a DEALLOCATE statement.
42	**severe (42): No such device** FOR$IOS_NO_SUCDEV. A pathname included an invalid or unknown device name when an OPEN operation was attempted.
43	**severe (43): File name specification error** FOR$IOS_FILNAMSPE. A pathname or file name given to an OPEN or INQUIRE statement was not acceptable to the Intel Fortran RTL I/O system.
44	**severe (44): Inconsistent record type** FOR$IOS_INCRECTYP. The RECORDTYPE value in an OPEN statement did not match the record type attribute of the existing file that was opened.
45	**severe (45): Keyword value error in OPEN statement** FOR$IOS_KEYVALERR. An improper value was specified for an OPEN or CLOSE statement specifier requiring a value.
46	**severe (46): Inconsistent OPEN/CLOSE parameters**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_INCOPECLO. Specifications in an OPEN or CLOSE statement were inconsistent. Some invalid combinations follow: • READONLY or ACTION='READ' with STATUS='NEW' or STATUS='SCRATCH' • READONLY with STATUS='REPLACE', ACTION='WRITE', or ACTION='READWRITE' • ACCESS='APPEND' with READONLY, ACTION='READ', STATUS='NEW', or STATUS='SCRATCH' • DISPOSE='SAVE', 'PRINT', or 'SUBMIT' with STATUS='SCRATCH' • DISPOSE='DELETE' with READONLY • CLOSE statement STATUS='DELETE' with OPEN statement READONLY • ACCESS='DIRECT' with POSITION='APPEND' or 'ASIS'
47	**severe (47): Write to READONLY file** FOR$IOS_WRIREAFIL. A write operation was attempted to a file that was declared ACTION='READ' or READONLY in the OPEN statement that is currently in effect.
48	**severe (48): Invalid argument to Fortran Run-Time Library** FOR$IOS_INVARGFOR. The compiler passed an invalid or improperly coded argument to the Intel Fortran RTL. This can occur if the compiler is newer than the RTL in use.
51	**severe (51): Inconsistent file organization** FOR$IOS_INCFILORG. The file organization specified in an OPEN statement did not match the organization of the existing file.
53	**severe (53): No current record** FOR$IOS_NO_CURREC. Attempted to execute a REWRITE statement to rewrite a record when the current record was undefined. To define the current record, execute a successful READ statement. You can optionally perform an INQUIRE statement on the logical unit after the READ statement and before the REWRITE statement. No other operations on the logical unit may be performed between the READ and REWRITE statements.
55	**severe (55): DELETE error** FOR$IOS_DELERR. An error condition was detected by the Intel Fortran RTL I/O system during execution of a DELETE statement.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
57	**severe (57): FIND error** FOR$IOS_FINERR. The Intel Fortran RTL I/O system detected an error condition during execution of a FIND statement.
58[1]	**info (58): Format syntax error at or near xx** FOR$IOS_FMTSYN. Check the statement containing *xx*, a character substring from the format string, for a format syntax error. For more information, see the FORMAT statement.
59	**severe (59): List-directed I/O syntax error** FOR$IOS_LISIO_SYN. The data in a list-directed input record had an invalid format, or the type of the constant was incompatible with the corresponding variable. The value of the variable was unchanged. Note: The ERR transfer is taken after completion of the I/O statement for error number 59. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
60	**severe (60): Infinite format loop** FOR$IOS_INFFORLOO. The format associated with an I/O statement that included an I/O list had no field descriptors to use in transferring those values.
61	**severe or info(61): Format/variable-type mismatch** FOR$IOS_FORVARMIS. An attempt was made either to read or write a real variable with an integer field descriptor (I, L, O, Z, B), or to read or write an integer or logical variable with a real field descriptor (D, E, or F). To suppress this error message, see the description of /check:noformat. Note: The severity depends on the -check keywords or /check:keywords option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
62	**severe (62): Syntax error in format** FOR$IOS_SYNERRFOR. A syntax error was encountered while the RTL was processing a format stored in an array or character variable.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
63	**error or info(63): Output conversion error** FOR$IOS_OUTCONERR. During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. When this situation is encountered, the overflowed field is filled with asterisks to indicate the error in the output record. If no ERR address has been defined for this error, the program continues after the error message is displayed. To suppress this error message, see the description of /check:nooutput_conversion. Note: The severity depends on the -check keywords or /check:keywords option used during the compilation command. The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
64	**severe (64): Input conversion error** FOR$IOS_INPCONERR. During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable was set to zero. Note: The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
65	**error (65): Floating invalid** FOR$IOS_FLTINV. During an arithmetic operation, the floating-point values used in a calculation were invalid for the type of operation requested or invalid exceptional values. For example, the error can occur if you request a log of the floating-point values 0.0 or a negative number. For certain arithmetic expressions, specifying the /check:nopower option can suppress this message.
66	**severe (66): Output statement overflows record** FOR$IOS_OUTSTAOVE. An output statement attempted to transfer more data than would fit in the maximum record size.
67	**severe (67): Input statement requires too much data** FOR$IOS_INPSTAREQ. Attempted to read more data than exists in a record with an unformatted `READ` statement or with a formatted sequential `READ` statement from a file opened with a PAD specifier value of 'NO'.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
68	**severe (68): Variable format expression value error** FOR$IOS_VFEVALERR. The value of a variable format expression was not within the range acceptable for its intended use; for example, a field width was less than or equal to zero. A value of 1 was assumed, except for a P edit descriptor, for which a value of zero was assumed. Note: The ERR transfer is taken after completion of the I/O statement for error numbers 61, 63, 64, and 68. The resulting file status and record position are the same as if no error had occurred. However, other I/O errors take the ERR transfer as soon as the error is detected, so file status and record position are undefined.
69[1]	**error (69): Process interrupted (SIGINT)** FOR$IOS_SIGINT. The process received the signal SIGINT. Determine source of this interrupt signal (described in `signal(3)`).
70[1]	**severe (70): Integer overflow** FOR$IOS_INTOVF. During an arithmetic operation, an integer value exceeded byte, word, or longword range. The result of the operation was the correct low-order part. Consider specifying a larger integer data size (modify source program or, for an INTEGER declaration, possibly use the /integer-size:size option).
71[1]	**severe (71): Integer divide by zero** FOR$IOS_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. The result of the operation was set to the dividend, which is equivalent to division by 1.
72[1]	**error (72): Floating overflow** FOR$IOS_FLTOVF. During an arithmetic operation, a floating-point value exceeded the largest representable value for that data type. See Data Representation for ranges of the various data types.
73[1]	**error (73): Floating divide by zero** FOR$IOS_FLTDIV. During a floating-point arithmetic operation, an attempt was made to divide by zero.
74[1]	**error (74): Floating underflow**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_FLTUND. During an arithmetic operation, a floating-point value became less than the smallest finite value for that data type. Depending on the values of the /fpe:*n* option, the underflowed result was either set to zero or allowed to gradually underflow. See the Data Representation for ranges of the various data types.
75[1]	**error (75): Floating point exception** FOR$IOS_SIGFPE. A floating-point exception occurred. Possible causes include: • Division by zero • Overflow • An invalid operation, such as subtraction of infinite values, multiplication of zero by infinity without signs), division of zero by zero or infinity by infinity • Conversion of floating-point to fixed-point format when an overflow prevents conversion
76[1]	**error (76): IOT trap signal** FOR$IOS_SIGIOT. Core dump file created. Examine core dump for possible cause of this IOT signal.
77[1]	**severe (77): Subscript out of range** FOR$IOS_SUBRNG. An array reference was detected outside the declared array bounds.
78[1]	**error (78): Process killed** FOR$IOS_SIGTERM. The process received a signal requesting termination of this process. Determine the source of this software termination signal.
79[1]	**error (79): Process quit** FOR$IOS_SIGQUIT. The process received a signal requesting termination of itself. Determine the source of this quit signal.
95[1]	**info (95): Floating-point conversion failed** FOR$IOS_FLOCONFAI. The attempted unformatted read or write of nonnative floating-point data failed because the floating-point value: • Exceeded the allowable maximum value for the equivalent native format and was set equal to infinity (plus or minus)

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• Was infinity (plus or minus) and was set to infinity (plus or minus) • Was invalid and was set to not a number (NaN) Very small numbers are set to zero (0). This error could be caused by the specified nonnative floating-point format not matching the floating-point format found in the specified file. Check the following: • The correct file was specified. • The record layout matches the format Intel Fortran is expecting. • The ranges for the data being used (see Data Representation). • The correct nonnative floating-point data format was specified (see Supported Native and Nonnative Numeric Formats).
96	**info (96): F_UFMTENDIAN environment variable was ignored:erroneous syntax** FOR$IOS_UFMTENDIAN. Syntax for specifying whether little endian or big endian conversion is performed for a given Fortran unit was incorrect. Even though the program will run, the results might not be correct if you do not change the value of F_UFMTENDIAN. For correct syntax, see Environment Variable F_UFMTENDIAN Method.
108	**Severe (108): Cannot stat file** FOR$IOS_CANSTAFILE. Make sure correct file and unit were specified.
120	**severe (120): Operation requires seek ability** FOR$IOS_OPEREQSEE. Attempted an operation on a file that requires the ability to perform seek operations on that file. Make sure the correct unit, directory path, and file were specified.
134	**No associated message** Program was terminated internally through abort().
138[1]	**severe (138): Array index out of bounds** FOR$IOS_BRK_RANGE. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.
139[1]	**severe: (139): Array index out of bounds for index nn**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_BRK_RANGE2. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.
140[1]	**error (140): Floating inexact** FOR$IOS_FLTINE. A floating-point arithmetic or conversion operation gave a result that differs from the mathematically exact result. This trap is reported if the rounded result of an IEEE operation is not exact.
144[1]	**severe (144): Reserved operand** FOR$IOS_ROPRAND. The Intel Fortran RTL encountered a reserved operand while executing your program. Please report the problem to Intel.
145[1]	**severe (145): Assertion error** FOR$IOS_ASSERTERR. The Intel Fortran RTL encountered an assertion error. Please report the problem to Intel.
146[1]	**severe (146): Null pointer error** FOR$IOS_NULPTRERR. Attempted to use a pointer that does not contain an address. Modify the source program, recompile, and relink.
147[1]	**severe (147): Stack overflow** FOR$IOS_STKOVF. The Intel Fortran RTL encountered a stack overflow while executing your program.
148[1]	**severe (148): String length error** FOR$IOS_STRLENERR. During a string operation, an integer value appears in a context where the value of the integer is outside the permissible string length range. Recompile with the /check:bounds option.
149[1]	**severe (149): Substring error** FOR$IOS_SUBSTRERR. An array subscript is outside the dimensioned boundaries of an array. Recompile with the /check:bounds option.
150[1]	**severe (150): Range error** FOR$IOS_RANGEERR. An integer value appears in a context where the value of the integer is outside the permissible range.
151[1]	**severe (151): Allocatable array is already allocated**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_INVREALLOC. An allocatable array must not already be allocated when you attempt to allocate it. You must deallocate the array before it can again be allocated. Note: This error can be returned by STAT in an ALLOCATE statement.
152[1]	**severe (152): Unresolved contention for DEC Fortran RTL global resource** FOR$IOS_RESACQFAI. Failed to acquire an Intel Fortran RTL global resource for a reentrant routine. For a multithreaded program, the requested global resource is held by a different thread in your program. For a program using asynchronous handlers, the requested global resource is held by the calling part of the program (such as main program) and your asynchronous handler attempted to acquire the same global resource.
153[1]	**severe (153): Allocatable array or pointer is not allocated** FOR$IOS_INVDEALLOC. A Fortran 90 allocatable array or pointer must already be allocated when you attempt to deallocate it. You must allocate the array or pointer before it can again be deallocated. Note: This error can be returned by STAT in a DEALLOCATE statement.
154[1]	**severe(154): Array index out of bounds** FOR$IOS_RANGE. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.
155[1]	**severe(155): Array index out of bounds for index nn** FOR$IOS_RANGE2. An array subscript is outside the dimensioned boundaries of that array. Recompile with the /check:bounds option set.
156[1]	**severe(156): GENTRAP code = hex dec** FOR$IOS_DEF_GENTRAP. The Intel Fortran RTL has detected an unknown GENTRAP code. The cause is most likely a software problem due to memory corruption, or software signalling an exception with an incorrect exception code. Try recompiling with the /check:bounds option set to see if that finds the problem.
157[1]	**severe(157): Program Exception - access violation** FOR$IOS_ACCVIO. The program tried to read from or write to a virtual address for which it does not have the appropriate access. Try recompiling with the /check:bounds option set, to see if the problem is an out-of-bounds memory reference or a argument mismatch that causes data to be treated as an address.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

Other causes of this error include:

- Mismatches in C vs. STDCALL calling mechanisms, causing the stack to become corrupted
- References to unallocated pointers
- Attempting to access a protected (for example, read-only) address

158[1] **severe(158): Program Exception - datatype misalignment**

FOR$IOS_DTYPE_MISALIGN. The Intel Fortran RTL has detected data that is not aligned on a natural boundary for the data type specified. For example, a REAL(8) data item aligned on natural boundaries has an address that is a multiple of 8. To ensure naturally aligned data, use the /align option.

This is an operating system error. See your operating system documentation for more information.

159[1] **severe(159): Program Exception - breakpoint**

FOR$IOS_PGM_BPT. The Intel Fortran RTL has encountered a breakpoint in the program.

This is an operating system error. See your operating system documentation for more information.

160[1] **severe(160): Program Exception - single step**

FOR$IOS_PGM_SS. A trace trap or other single-instruction mechanism has signaled that one instruction has been executed.

This is an operating system error. See your operating system documentation for more information.

161[1] **severe(161): Program Exception - array bounds exceeded**

FOR$IOS_PGM_BOUNDS. The program tried to access an array element that is outside the specified boundaries of the array. Recompile with the /check:bounds option set.

162[1] **severe(162): Program Exception - denormal floating-point operand**

FOR$IOS_PGM_DENORM. A floating-point arithmetic or conversion operation has a denormalized number as an operand. A denormalized number is smaller than the lowest value in the normal range for the data type specified. See Data Representation for ranges for floating-point types.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	Either locate and correct the source code causing the denormalized value or, if a denormalized value is acceptable, specify a different value for the /fpe compiler option to allow program continuation.
163[1]	**severe(163): Program Exception - floating stack check** FOR$IOS_PGM_FLTSTK. During a floating-point operation, the floating-point register stack on systems using IA-32 architecture overflowed or underflowed. This is a fatal exception. The most likely cause is calling a REAL function as if it were an INTEGER function or subroutine, or calling an INTEGER function or subroutine as if it were a REAL function. Carefully check that the calling code and routine being called agree as to how the routine is declared. If you are unable to resolve the issue, please send a problem report with an example to Intel.
164[1]	**severe(164): Program Exception - integer divide by zero** FOR$IOS_PGM_INTDIV. During an integer arithmetic operation, an attempt was made to divide by zero. Locate and correct the source code causing the integer divide by zero.
165[1]	**severe(165): Program Exception - integer overflow** FOR$IOS_PGM_INTOVF. During an arithmetic operation, an integer value exceeded the largest representable value for that data type. See Data Representation for ranges for INTEGER types.
166[1]	**severe(166): Program Exception - privileged instruction** FOR$IOS_PGM_PRIVINST. The program tried to execute an instruction whose operation is not allowed in the current machine mode. This is an operating system error. See your operating system documentation for more information.
167[1]	**severe(167): Program Exception - in page error** FOR$IOS_PGM_INPGERR. The program tried to access a page that was not present, so the system was unable to load the page. For example, this error might occur if a network connection was lost while trying to run a program over the network. This is an operating system error. See your operating system documentation for more information.
168[1]	**severe(168): Program Exception - illegal instruction**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_PGM_ILLINST. The program tried to execute an invalid instruction. This is an operating system error. See your operating system documentation for more information.
169[1]	**severe(169): Program Exception - noncontinuable exception** FOR$IOS_PGM_NOCONTEXCP. The program tried to continue execution after a noncontinuable exception occurred. This is an operating system error. See your operating system documentation for more information.
170[1]	**severe(170): Program Exception - stack overflow** FOR$IOS_PGM_STKOVF. The Intel Fortran RTL has detected a stack overflow while executing your program. See your Release Notes for information on how to increase stack size.
171[1]	**severe(171): Program Exception - invalid disposition** FOR$IOS_PGM_INVDISP. An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language should never encounter this exception. This is an operating system error. See your operating system documentation for more information.
172[1]	**severe(172): Program Exception - exception code = hex dec** FOR$IOS_PGM_EXCP_CODE. The Intel Fortran RTL has detected an unknown exception code. This is an operating system error. See your operating system documentation for more information.
173[1]	**severe(173): A pointer passed to DEALLOCATE points to an array that cannot be deallocated** FOR$IOS_INVDEALLOC2. A pointer that was passed to DEALLOCATE pointed to an explicit array, an array slice, or some other type of memory that could not be deallocated in a `DEALLOCATE` statement. Only whole arrays previous allocated with an `ALLOCATE` statement may be validly passed to DEALLOCATE. Note: This error can be returned by STAT in a DEALLOCATE statement.
174[1]	**severe (174): SIGSEGV, message-text**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_SIGSEGV. One of two possible messages occurs for this error number: • `severe (174): SIGSEGV, segmentation fault occurred` This message indicates that the program attempted an invalid memory reference. Check the program for possible errors. • `severe (174): SIGSEGV, possible program stack overflow occurred` The following explanatory text also appears: `Program requirements exceed current stacksize resource limit.`
175[1]	**severe(175): DATE argument to DATE_AND_TIME is too short (LEN=n), required LEN=8** FOR$IOS_SHORTDATEARG. The number of characters associated with the DATE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 8 characters in length. Verify that the TIME and ZONE arguments also meet their minimum lengths.
176[1]	**severe(176): TIME argument to DATE_AND_TIME is too short (LEN=n), required LEN=10** FOR$IOS_SHORTTIMEARG. The number of characters associated with the TIME argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 10 characters in length. Verify that the DATE and ZONE arguments also meet their minimum lengths.
177[1]	**severe(177): ZONE argument to DATE_AND_TIME is too short (LEN=n), required LEN=5** FOR$IOS_SHORTZONEARG. The number of characters associated with the ZONE argument to the DATE_AND_TIME intrinsic was shorter than the required length. You must increase the number of characters passed in for this argument to be at least 5 characters in length. Verify that the DATE and TIME arguments also meet their minimum lengths.
178[1]	**severe(178): Divide by zero** FOR$IOS_DIV. A floating-point or integer divide-by-zero exception occurred.
179[1]	**severe(179): Cannot allocate array - overflow on array size calculation**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_ARRSIZEOVF. An attempt to dynamically allocate storage for an array failed because the required storage size exceeds addressable memory. Note: This error can be returned by STAT in an ALLOCATE statement.
256	**severe (256): Unformatted I/O to unit open for formatted transfers** FOR$IOS_UNFIO_FMT. Attempted unformatted I/O to a unit where the OPEN statement (FORM specifier) indicated the file was formatted. Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains unformatted data, specify FORM='UNFORMATTED'in the OPEN statement. Otherwise, if appropriate, use formatted I/O (such as list-directed or namelist I/O).
257	**severe (257): Formatted I/O to unit open for unformatted transfers** FOR$IOS_FMTIO_UNF. Attempted formatted I/O (such as list-directed or namelist I/O) to a unit where the OPEN statement indicated the file was unformatted (FORM specifier). Check that the correct unit (file) was specified. If the FORM specifier was not present in the OPEN statement and the file contains formatted data, specify FORM='FORMATTED' in the OPEN statement. Otherwise, if appropriate, use unformatted I/O.
259	**severe (259): Sequential-access I/O to unit open for direct access** FOR$IOS_SEQIO_DIR. The OPEN statement for this unit number specified direct access and the I/O statement specifies sequential access. Check the OPEN statement and make sure the I/O statement uses the correct unit number and type of access.
264	**severe (264): operation requires file to be on disk or tape** FOR$IOS_OPEREQDIS. Attempted to use a BACKSPACE statement on such devices as a terminal.
265	**severe (265): operation requires sequential file organization and access** FOR$IOS_OPEREQSEQ. Attempted to use a BACKSPACE statement on a file whose organization was not sequential or whose access was not sequential. A BACKSPACE statement can only be used for sequential files opened for sequential access.
266[1]	**error (266): Fortran abort routine called** FOR$IOS_PROABOUSE. The program called the abort routine to terminate itself.
268[1]	**severe (268): End of record during read**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_ENDRECDUR. An end-of-record condition was encountered during execution of a nonadvancing I/O `READ` statement that did not specify the EOR branch specifier.
296[1]	**info(296): nn floating inexact traps** FOR$IOS_FLOINEEXC. The total number of floating-point inexact data traps encountered during program execution was *nn*. This summary message appears at program completion.
297[1]	**info (297): nn floating invalid traps** FOR$IOS_FLOINVEXC. The total number of floating-point invalid data traps encountered during program execution was *nn*. This summary message appears at program completion.
298[1]	**info (298): nn floating overflow traps** FOR$IOS_FLOOVFEXC. The total number of floating-point overflow traps encountered during program execution was *nn*. This summary message appears at program completion.
299[1]	**info (299): nn floating divide-by-zero traps** FOR$IOS_FLODIV0EXC. The total number of floating-point divide-by-zero traps encountered during program execution was *nn*. This summary message appears at program completion.
300[1]	**info (300): nn floating underflow traps** FOR$IOS_FLOUNDEXC. The total number of floating-point underflow traps encountered during program execution was *nn*. This summary message appears at program completion.
540	**severe (540): Array or substring subscript expression out of range** FOR$IOS_F6096. An expression used to index an array was smaller than the lower dimension bound or larger than the upper dimension bound.
541	**severe (541): CHARACTER substring expression out of range** FOR$IOS_F6097. An expression used to index a character substring was illegal.
542	**severe (542): Label not found in assigned GOTO list**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6098. The label assigned to the integer-variable name was not specified in the label list of the assigned GOTO statement.
543	**severe (543): INTEGER arithmetic overflow**
	FOR$IOS_F6099. This error occurs whenever integer arithmetic results in overflow.
544	**severe (544): INTEGER overflow on input**
	FOR$IOS_F6100. An integer item exceeded the legal size limits.
	An INTEGER (1) item must be in the range -127 to 128. An INTEGER (2) item must be in the range -32,767 to 32,768. An INTEGER (4) item must be in the range -2,147,483,647 to 2,147,483,648.
545	**severe (545): Invalid INTEGER**
	FOR$IOS_F6101. Either an illegal character appeared as part of an integer, or a numeric character larger than the radix was used in an alternate radix specifier.
546	**severe (546): REAL indefinite (uninitialized or previous error)**
	FOR$IOS_F6102. An invalid real number was read from a file, an internal variable, or the console. This can happen if an invalid number is generated by passing an illegal argument to an intrinsic function -- for example, SQRT(-1) or ASIN(2). If the invalid result is written and then later read, the error will be generated.
547	**severe (547): Invalid REAL**
	FOR$IOS_F103. An illegal character appeared as part of a real number.
548	**severe (548): REAL math overflow**
	FOR$IOS_F6104. A real value was too large. Floating-point overflows in either direct or emulated mode generate NaN (Not-A-Number) exceptions, which appear in the output field as asterisks (*) or the letters NAN.
550	**severe (550): INTEGER assignment overflow**
	FOR$IOS_F6106. This error occurs when assignment to an integer is out of range.
551	**severe (551): Formatted I/O not consistent with OPEN options**
	FOR$IOS_F6200. The program tried to perform formatted I/O on a unit opened with FORM='UNFORMATTED' or FORM='BINARY'.
552	**severe (552): List-directed I/O not consistent with OPEN options**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6201. The program tried to perform list-directed I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL'.
553	**severe (553): Terminal I/O not consistent with OPEN options** FOR$IOS_F6202. When a special device such as CON, LPT1, or PRN is opened in an OPEN statement, its access must be sequential and its format must be either formatted or binary. By default ACCESS='SEQUENTIAL' and FORM='FORMATTED' in OPEN statements. To generate this error the device's OPEN statement must contain an option not appropriate for a terminal device, such as ACCESS='DIRECT' or FORM='UNFORMATTED'.
554	**severe (554): Direct I/O not consistent with OPEN options** FOR$IOS_F6203. A REC= option was included in a statement that transferred data to a file that was opened with the ACCESS='SEQUENTIAL' option.
555	**severe (555): Unformatted I/O not consistent with OPEN options** FOR$IOS_F6204. If a file is opened with FORM='FORMATTED', unformatted or binary data transfer is prohibited.
556	**severe (556): A edit descriptor expected for CHARACTER** FOR$IOS_F6205. The A edit descriptor was not specified when a character data item was read or written using formatted I/O.
557	**severe (557): E, F, D, or G edit descriptor expected for REAL** FOR$IOS_F6206. The E, F, D, or G edit descriptor was not specified when a real data item was read or written using formatted I/O.
558	**severe (558): I edit descriptor expected for INTEGER** FOR$IOS_F6207. The I edit descriptor was not specified when an integer data item was read or written using formatted I/O.
559	**severe (559): L edit descriptor expected for LOGICAL** FOR$IOS_F6208. The L edit descriptor was not specified when a logical data item was read or written using formatted I/O.
560	**severe (560): File already open: parameter mismatch**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6209. An OPEN statement specified a connection between a unit and a filename that was already in effect. In this case, only the BLANK= option can have a different setting.
561	**severe (561): Namelist I/O not consistent with OPEN options** FOR$IOS_F6210. The program tried to perform namelist I/O on a file that was not opened with FORM='FORMATTED' and ACCESS='SEQUENTIAL.'
562	**severe (562): IOFOCUS option illegal with non-window unit** FOR$IOS_F6211. IOFOCUS was specified in an OPEN or INQUIRE statement for a non-window unit. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.
563	**severe (563): IOFOCUS option illegal without QuickWin** FOR$IOS_F6212. IOFOCUS was specified in an OPEN or INQUIRE statement for a non-QuickWin application. The IOFOCUS option can only be used when the unit opened or inquired about is a QuickWin child window.
564	**severe (564): TITLE illegal with non-window unit** FOR$IOS_F6213. TITLE was specified in an OPEN or INQUIRE statement for a non-window unit. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.
565	**severe (565): TITLE illegal without QuickWin** FOR$IOS_F6214. TITLE was specified in an OPEN or INQUIRE statement for a non-QuickWin application. The TITLE option can only be used when the unit opened or inquired about is a QuickWin child window.
566	**severe (566): KEEP illegal for scratch file** FOR$IOS_F6300. STATUS='KEEP' was specified for a scratch file; this is illegal because scratch files are automatically deleted at program termination.
567	**severe (567): SCRATCH illegal for named file** FOR$IOS_F6301. STATUS='SCRATCH' should not be used in a statement that includes a filename.
568	**severe (568): Multiple radix specifiers**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6302. More than one alternate radix for numeric I/O was specified. F6302 can indicate an error in spacing or a mismatched format for data of different radices.
569	**severe (569): Illegal radix specifier** FOR$IOS_F6303. A radix specifier was not between 2 and 36, inclusive. Alternate radix constants must be of the form n#ddd... where n is a radix from 2 to 36 inclusive and ddd... are digits with values less than the radix. For example, 3#12 and 34#7AX are valid constants with valid radix specifiers. 245#7A and 39#12 do not have valid radix specifiers and generate error 569 if input.
570	**severe (570): Illegal STATUS value** FOR$IOS_F6304. An illegal value was used with the STATUS option. STATUS accepts the following values: • 'KEEP' or 'DELETE' when used with CLOSE statements • 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN' when used with OPEN statements
571	**severe (571): Illegal MODE value** FOR$IOS_F6305. An illegal value was used with the MODE option. MODE accepts the values 'READ', 'WRITE', or 'READWRITE'.
572	**severe (572): Illegal ACCESS value** FOR$IOS_F6306. An illegal value was used with the ACCESS option. ACCESS accepts the values 'SEQUENTIAL' and 'DIRECT'.
573	**severe (573): Illegal BLANK value** FOR$IOS_F6307. An illegal value was used with the BLANK option. BLANK accepts the values 'NULL' and 'ZERO'.
574	**severe (574): Illegal FORM value** FOR$IOS_F6308. An illegal value was used with the FORM option. FORM accepts the following values: 'FORMATTED', 'UNFORMATTED', and 'BINARY'.
575	**severe (575): Illegal SHARE value** FOR$IOS_F6309. An illegal value was used with the SHARE option.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	SHARE accepts the values 'COMPAT', 'DENYRW', 'DENYWR', 'DENYRD', and 'DENYNONE'.
577	**severe (577): Illegal record number** FOR$IOS_F6311. An invalid number was specified as the record number for a direct-access file. The first valid record number for direct-access files is 1.
578	**severe (578): No unit number associated with *** FOR$IOS_F6312. In an INQUIRE statement, the NUMBER option was specified for the file associated with * (console).
580	**severe (580): Illegal unit number** FOR$IOS_F6314. An illegal unit number was specified. Legal unit numbers can range from 0 through 2**31-1, inclusive.
581	**severe (581): Illegal RECL value** FOR$IOS_F6315. A negative or zero record length was specified for a direct file. The smallest valid record length for direct files is 1.
582	**severe (582): Array already allocated** FOR$IOS_F6316. The program attempted to `ALLOCATE` an already allocated array.
583	**severe (583): Array size zero or negative** FOR$IOS_F6317. The size specified for an array in an ALLOCATE statement must be greater than zero.
584	**severe (584): Non-HUGE array exceeds 64K** FOR$IOS_F6318.
585	**severe (585): Array not allocated** FOR$IOS_F6319. The program attempted to `DEALLOCATE` an array that was never allocated.
586	**severe (586): BACKSPACE illegal on terminal device** FOR$IOS_F6400. A BACKSPACE statement specified a unit connected to a terminal device such as a terminal or printer.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
587	**severe (587): EOF illegal on terminal device** FOR$IOS_F6401. An EOF intrinsic function specified a unit connected to a terminal device such as a terminal or printer.
588	**severe (588): ENDFILE illegal on terminal device** FOR$IOS_F6402. An ENDFILE statement specified a unit connected to a terminal device such as a terminal or printer.
589	**severe (589): REWIND illegal on terminal device** FOR$IOS_F6403. A `REWIND` statement specified a unit connected to a terminal device such as a terminal or printer.
590	**severe (590): DELETE illegal for read-only file** FOR$IOS_F6404. A CLOSE statement specified STATUS='DELETE' for a read-only file.
591	**severe (591): External I/O illegal beyond end of file** FOR$IOS_F6405. The program tried to access a file after executing an ENDFILE statement or after it encountered the end-of-file record during a read operation. A BACKSPACE, REWIND, or OPEN statement must be used to reposition the file before execution of any I/O statement that transfers data.
592	**severe (592): Truncation error: file closed** FOR$IOS_F6406.
593	**severe (593): Terminal buffer overflow** FOR$IOS_F6407. More than 131 characters were input to a record of a unit connected to the terminal (keyboard). Note that the operating system may impose additional limits on the number of characters that can be input to the terminal in a single record.
594	**severe (594): Comma delimiter disabled after left repositioning** FOR$IOS_F6408. If you have record lengths that exceed the buffer size associated with the record, (for instance, the record is a file with the buffer set by BLOCKSIZE in the OPEN statement), either you should not do left tabbing within the record, or you should not use commas as field delimiters. This is because commas are disabled as input field delimiters if left tabbing leaves the record positioned in a previous buffer.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

For example, consider you have a file LONG.DAT that is one continuous record with data fields separated by commas. You then set the buffer associated with the file to 512 bytes, read more than one buffer size of data, tab left to data in the previous buffer, and attempt to read further data, as follows:

```
    INTEGER value(300)
   OPEN (1, FILE = 'LONG.DAT', BLOCKSIZE = 512)s
   READ (1, 100) (value(i), i = 1, 300)s
100 FORMAT (290I2,TL50,10I2)
```

In this case, error 594 occurs.

599 **severe (599): File already connected to a different unit**

FOR$IOS_F6413. The program tried to connect an already connected file to a new unit.

A file can be connected to only one unit at a time.

600 **severe (600): Access not allowed**

FOR$IOS_F6414.

This error can be caused by one of the following:

- The filename specified in an OPEN statement was a directory.
- An OPEN statement tried to open a read-only file for writing.
- The file was opened with SHARE='DENYRW' by another process.

601 **severe (601): File already exists**

FOR$IOS_F6415. An OPEN statement specified STATUS='NEW' for a file that already exists.

602 **severe (602): File not found**

FOR$IOS_F6416. An OPEN statement specified STATUS='OLD' for a specified file or a directory path that does not exist.

603 **severe (603): Too many open files**

FOR$IOS_F6417. The program exceeded the number of open files the operating system allows.

604 **severe (604): Too many units connected**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6418. The program exceeded the number of units that can be connected at one time. Units are connected with the OPEN statement.
605	**severe (605): Illegal structure for unformatted file** FOR$IOS_F6419. The file was opened with FORM='UNFORMATTED' and ACCESS='SEQUENTIAL', but its internal physical structure was incorrect or inconsistent. Possible causes: the file was created in another mode or by a non-Fortran program.
606	**severe (606): Unknown unit number** FOR$IOS_F6420. A statement such as BACKSPACE or ENDFILE specified a file that had not yet been opened. (The READ and WRITE statements do not cause this problem because they prompt you for a file if the file has not been opened yet.)
607	**severe (607): File read-only or locked against writing** FOR$IOS_F6421. The program tried to transfer data to a file that was opened in read-only mode or locked against writing. The error message may indicate a CLOSE error when the fault is actually coming from WRITE. This is because the error is not discovered until the program tries to write buffered data when it closes the file.
608	**severe (608): No space left on device** FOR$IOS_F6422. The program tried to transfer data to a file residing on a device (such as a hard disk) that was out of storage space.
609	**severe (609): Too many threads** FOR$IOS_F6423. Too many threads were active simultaneously. At most, 32 threads can be active at one time. Close any unnecessary processes or child windows within your application.
610	**severe (610): Invalid argument** FOR$IOS_F6424.
611	**severe (611): BACKSPACE illegal for SEQUENTIAL write-only files** FOR$IOS_F6425. The BACKSPACE statement is not allowed in files opened with MODE='WRITE' (write-only status) because BACKSPACE requires reading the previous record in the file to provide positioning.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	Resolve the problem by giving the file read access or by avoiding the BACKSPACE statement. Note that the REWIND statement is valid for files opened as write-only.
612	**severe (612): File not open for reading or file locked** FOR$IOS_F6500. The program tried to read from a file that was not opened for reading or was locked.
613	**severe (613): End of file encountered** FOR$IOS_F6501. The program tried to read more data than the file contains.
614	**severe (614): Positive integer expected in repeat field** FOR$IOS_F6502. When the *i***c* form is used in list-directed input, the *i* must be a positive integer. For example, consider the following statement: `READ(*,*) a, b` Input 2*56.7 is accepted, but input 2.1*56.7 returns error 614.
615	**severe (615): Multiple repeat field** FOR$IOS_F6503. In list-directed input of the form *i***c*, an extra repeat field was used. For example, consider the following: `READ(*,*) I, J, K` Input of 2*1*3 returns this error. The 2*1 means send two values, each 1; the *3 is an error.
616	**severe (616): Invalid number in input** FOR$IOS_F6504. Some of the values in a list-directed input record were not numeric. For example, consider the following: `READ(*,*) I, J` The preceding statement would cause this error if the input were: 123 'abc'.
617	**severe (617): Invalid string in input** FOR$IOS_F6505. A string item was not enclosed in single quotation marks.
618	**severe (618): Comma missing in COMPLEX input** FOR$IOS_F6506. When using list-directed input, the real and imaginary components of a complex number were not separated by a comma.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
619	**severe (619): T or F expected in LOGICAL read** FOR$IOS_F6507. The wrong format was used for the input field for logical data. The input field for logical data consists of optional blanks, followed by an optional decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field, so that .TRUE. and .FALSE. are acceptable input forms.
620	**severe (620): Too many bytes read from unformatted record** FOR$IOS_F6508. The program tried to read more data from an unformatted file than the current record contained. If the program was reading from an unformatted direct file, it tried to read more than the fixed record length as specified by the RECL option. If the program was reading from an unformatted sequential file, it tried to read more data than was written to the record.
621	**severe (621): H or apostrophe edit descriptor illegal on input** FOR$IOS_F6509. Hollerith (H) or apostrophe edit descriptors were encountered in a format used by a READ statement.
622	**severe (622): Illegal character in hexadecimal input** FOR$IOS_F6510. The input field contained a character that was not hexadecimal. Legal hexadecimal characters are 0 - 9 and A - F.
623	**severe (623): Variable name not found** FOR$IOS_F6511. A name encountered on input from a namelist record is not declared in the corresponding NAMELIST statement.
624	**severe (624): Invalid NAMELIST input format** FOR$IOS_F6512. The input record is not in the correct form for namelist input.
625	**severe (625): Wrong number of array dimensions** FOR$IOS_F6513. In namelist input, an array name was qualified with a different number of subscripts than its declaration, or a non-array name was qualified.
626	**severe (626): Array subscript exceeds allocated area** FOR$IOS_F6514. A subscript was specified in namelist input which exceeded the declared dimensions of the array.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
627	**severe (627): Invalid subrange in NAMELIST input** FOR$IOS_F6515. A character item in namelist input was qualified with a subrange that did not meet the requirement that 1 <= e1 <= e2 <= len (where "len" is the length of the character item, "e1" is the leftmost position of the substring, and "e2" is the rightmost position of the substring).
628	**severe (628): Substring range specified on non-CHARACTER item** FOR$IOS_F6516. A non-CHARACTER item in namelist input was qualified with a substring range.
629	**severe (629): Internal file overflow** FOR$IOS_F6600. The program either overflowed an internal-file record or tried to write to a record beyond the end of an internal file.
630	**severe (630): Direct record overflow** FOR$IOS_F6601. The program tried to write more than the number of bytes specified in the RECL option to an individual record of a direct-access file.
631	**severe (631):Numeric field bigger than record size** FOR$IOS_F6602. The program tried to write a noncharacter item across a record boundary in list-directed or namelist output. Only character constants can cross record boundaries.
632	**severe (632): Heap space limit exceeded** FOR$IOS_F6700. The program ran out of heap space. The ALLOCATE statement and various internal functions allocate memory from the heap. This error will be generated when the last of the heap space is used up.
633	**severe (633): Scratch file name limit exceeded** FOR$IOS_F6701. The program exhausted the template used to generate unique scratch-file names. The maximum number of scratch files that can be open at one time is 26.
634	**severe (634): D field exceeds W field in ES edit descriptor** FOR$IOS_F6970. The specified decimal length D exceeds the specified total field width W in an ES edit descriptor.
635	**severe (635): D field exceeds W field in EN edit descriptor**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6971. The specified decimal length D exceeds the specified total field width W in an EN edit descriptor.
636	**severe (636): Exponent of 0 not allowed in format** FOR$IOS_F6972.

637	**severe (637): Integer expected in format**
	FOR$IOS_F6980. An edit descriptor lacked a required integer value. For example, consider the following:
	`WRITE(*, 100) I, J`
	`100 FORMAT (I2, TL, I2)`
	The preceding code will cause this error because an integer is expected after TL.

638	**severe (638): Initial left parenthesis expected in format** FOR$IOS_F6981. A format did not begin with a left parenthesis (( ).
639	**severe (639): Positive integer expected in format** FOR$IOS_F6982. A zero or negative integer value was used in a format. Negative integer values can appear only with the P edit descriptor. Integer values of 0 can appear only in the *d* and *m* fields of numeric edit descriptors.
640	**severe (640): Repeat count on nonrepeatable descriptor** FOR$IOS_F6983. One or more BN, BZ, S, SS, SP, T, TL, TR, /, $, :, or apostrophe (') edit descriptors had repeat counts associated with them.
641	**severe (641): Integer expected preceding H, X, or P edit descriptor** FOR$IOS_F6984. An integer did not precede a (nonrepeatable) H, X, or P edit descriptor. The correct formats for these descriptors are *n*H, *n*X, and *k*P, respectively, where *n* is a positive integer and *k* is an optionally signed integer.
642	**severe (642): N or Z expected after B in format** FOR$IOS_F6985. To control interpretation of embedded and trailing blanks within numeric input fields, you must specify BN (to ignore them) or BZ (to interpret them as zeros).
643	**severe (643): Format nesting limit exceeded**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6986. More than 16 sets of parentheses were nested inside the main level of parentheses in a format.
644	**severe (644): '.' expected in format**
	FOR$IOS_F6987. No period appeared between the *w* and *d* fields of a D, E, F, or G edit descriptor.
645	**severe (645): Unexpected end of format**
	FOR$IOS_F6988. An incomplete format was used.
	Improperly matched parentheses, an unfinished Hollerith (H) descriptor, or another incomplete descriptor specification can cause this error.
646	**severe (646): Unexpected character in format**

	FOR$IOS_F6989. A character that cannot be interpreted as part of a valid edit descriptor was used in a format. For example, consider the following:
	`WRITE(*, 100) I, J`
	`100 FORMAT (I2, TL4.5, I2)`

	The code will generate this error because TL4.5 is not a valid edit descriptor. An integer must follow TL.
647	**severe (647): M field exceeds W field in I edit descriptor**
	FOR$IOS_F6990. In syntax I*w*.*m*, the value of *m* cannot exceed the value of *w*.
648	**severe (648): Integer out of range in format**
	FOR$IOS_F6991. An integer value specified in an edit descriptor was too large to represent as a 4-byte integer.
649	**severe (649): format not set by ASSIGN**
	FOR$IOS_F6992. The format specifier in a READ, WRITE, or PRINT statement was an integer variable, but an ASSIGN statement did not properly assign it the statement label of a FORMAT statement in the same program unit.
650	**severe (650): Separator expected in format**
	FOR$IOS_F6993. Within format specifications, edit descriptors must be separated by commas or slashes (/).
651	**severe (651): %c or $: nonstandard edit descriptor in format**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6994.
652	**severe (652): Z: nonstandard edit descriptor in format** FOR$IOS_F6995. Z is not a standard edit descriptor in format. If you want to transfer hexadecimal values, you must use the edit descriptor form Z*w*[.*m*], where *w* is the field width and *m* is the minimum number of digits that must be in the field (including leading zeros).
653	**severe (653): DOS graphics not supported under Windows NT** FOR$IOS_F6996.
654	**severe (654): Graphics error** FOR$IOS_F6997. An OPEN statement in which IOFOCUS was TRUE, either explicitly or by default, failed because the new window could not receive focus. The window handle may be invalid, or closed, or there may be a memory resource problem.
655	**severe (655): Using QuickWin is illegal in console application** FOR$IOS_F6998. A call to QuickWin from a console application was encountered during execution.
656	**severe (656): Illegal 'ADVANCE' value** FOR$IOS_F6999. The ADVANCE option can only take the values 'YES' and 'NO'. ADVANCE='YES' is the default. ADVANCE is a READ statement option.
657	**severe (657): DIM argument to SIZE out of range** FOR$IOS_F6702. The argument specified for DIM must be greater than or equal to 1, and less than or equal to the number of dimensions in the specified array. Consider the following: `i = SIZE (array, DIM = dim)` In this case, `1 <= dim <=` *n*, where *n* is the number of dimensions in array.
658	**severe (657): Undefined POINTER used as argument to ASSOCIATED function** FOR$IOS_F6703. A POINTER used as an argument to the ASSOCIATED function must be defined; that is, assigned to a target, allocated, or nullified.
659	**severe (659): Reference to uninitialized POINTER**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6704. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.
660	**severe (660): Reference to POINTER which is not associated** FOR$IOS_F6705. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.
661	**severe (661): Reference to uninitialized POINTER 'pointer'** FOR$IOS_F6706. Except in an assignment statement, a pointer must not be referenced until it has been initialized: assigned to a target, allocated or nullified.
662	**severe (662): reference to POINTER 'pointer' which is not associated** FOR$IOS_F6707. Except in an assignment statement and certain procedure references, a pointer must not be referenced until it has been associated: either assigned to a target or allocated.
663	**severe (663): Out of range: substring starting position 'pos' is less than 1** FOR$IOS_F6708. A substring starting position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.
664	**severe (664): Out of range: substring ending position 'pos' is greater than string length 'len'** FOR$IOS_F6709. A substring ending position must be a positive integer variable or expression that indicates a position in the string: at least 1 and no greater than the length of the string.
665	**severe (665): Subscript 'n' of 'str' (value 'val') is out of range ('first:last')** FOR$IOS_F6710. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.
666	**severe (666): Subscript 'n' of 'str' (value 'val') is out of range ('first:*')** FOR$IOS_F6711. The subscript for a substring within a string is not a valid string position: at least 1 and no greater than the length of the string.
667	**severe (667): VECTOR argument to PACK has incompatible character length** FOR$IOS_F6712. The character length of elements in the VECTOR argument to PACK is not the same as the character length of elements in the array to be packed.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
668	**severe (668): VECTOR argument to PACK is too small** FOR$IOS_F6713. The VECTOR argument to PACK must have at least as many elements as there are true elements in MASK (the array that controls packing).
669	**severe (669): SOURCE and PAD arguments to RESHAPE have different character lengths** FOR$IOS_F6714. The character length of elements in the SOURCE and PAD arguments to PACK must be the same.
670	**severe (670): Element 'n' of SHAPE argument to RESHAPE is negative** FOR$IOS_F6715. The SHAPE vector specifies the shape of the reshaped array. Since an array cannot have a negative dimension, SHAPE cannot have a negative element.
671	**severe (671): SOURCE too small for specified SHAPE in RESHAPE, and no PAD** FOR$IOS_F6716. If there is no PAD array, the SOURCE argument to RESHAPE must have enough elements to make an array of the shape specified by SHAPE.
672	**severe (672): Out of memory** FOR$IOS_F6717. The system ran out of memory while trying to make the array specified by RESHAPE. If possible, reset your virtual memory size through the Windows Control Panel, or close unneccessary applications and deallocate all allocated arrays that are no longer needed.
673	**severe (673): SHAPE and ORDER arguments to RESHAPE have different sizes ('size1' and 'size2')** FOR$IOS_F6718. ORDER specifies the order of the array dimensions given in SHAPE, and they must be vectors of the same size.
674	**severe (674): Element 'n' of ORDER argument to RESHAPE is out of range ('range')** FOR$IOS_F6719. The ORDER argument specifies the order of the dimensions of the reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array.
675	**severe (675): Value 'val' occurs twice in ORDER argument to RESHAPE**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6720. The ORDER vector specifies the order of the dimensions of the reshaped array, and it must be a permuted list of (1, 2, ..., *n*) where *n* is the highest dimension in the reshaped array. No dimension can occur twice.
676	**severe (676): Impossible nextelt overflow in RESHAPE** FOR$IOS_F6721.
677	**severe (677): Invalid value 'dim' for argument DIM for SPREAD of rank 'rank' source** FOR$IOS_F6722. The argument specified for DIM to SPREAD must be greater than or equal to 1, and less than or equal to one larger than the number of dimensions (rank) of SOURCE. Consider the following statement: `result = SPREAD (SOURCE= array, DIM = dim, NCOPIES = k)` In this case, 1 <= dim <= *n*+ 1, where *n*is the number of dimensions in array.
678	**severe (678): Complex zero raised to power zero** FOR$IOS_F6723. Zero of any type (complex, real, or integer) cannot be raised to zero power.
679	**severe (679): Complex zero raised to negative power** FOR$IOS_F6724. Zero of any type (complex, real, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.
680	**severe (680): Impossible error in NAMELIST input** FOR$IOS_F6725.
681	**severe (681):DIM argument to CSHIFT ('dim') is out of range** FOR$IOS_F6726. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <=DIM <= *n*, where *n*is the number of dimensions in the array to be shifted.
682	**severe (682): DIM argument ('dim') to CSHIFT is out of range (1:'n')** FOR$IOS_F6727. The optional argument DIM specifies the dimension along which to perform the circular shift, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array to be shifted. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in the array to be shifted.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
683	**severe (683): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in CSHIFT** FOR$IOS_F6728. The SHIFT argument to CSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension except the one being shifted along.
684	**severe (684): Internal error - bad arguments to CSHIFT_CA** FOR$IOS_F6729.
685	**severe (685): Internal error - bad arguments to CSHIFT_CAA** FOR$IOS_F6730.
686	**severe (686): DATE argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6731. The character DATE argument must have a length of at least 8 to contain the complete value.
687	**severe (687): TIME argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6732. The character TIME argument must have a length of at least 10 to contain the complete value.
688	**severe (688): ZONE argument to DATE_AND_TIME is too short (LEN='len')** FOR$IOS_F6733. The character ZONE argument must have a length of at least 5 to contain the complete value.
689	**severe (689): VALUES argument to DATE_AND_TIME is too small ('size' elements)** FOR$IOS_F6734. The integer VALUES argument must be a one-dimensional array with a size of at least 8 to hold all returned values.
690	**severe (690): Out of range: DIM argument to COUNT has value 'dim'** FOR$IOS_F6735. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions in MASK. That is, 1 <= DIM <= n, where n is the number of dimensions in MASK.
691	**severe (691): Out of range: DIM argument to COUNT has value 'dim' with MASK of rank 'rank'**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6736. The optional argument DIM specifies the dimension along which to count true elements of MASK, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in MASK. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in MASK.
692	**severe (692): Out of range: DIM argument to PRODUCT has value 'dim'**
	FOR$IOS_F6737. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be multiplied.
693	**severe (693): Out of range: DIM argument to PRODUCT has value 'dim' with ARRAY of rank 'rank'**
	FOR$IOS_F6738. The optional argument DIM specifies the dimension along which to compute the product of elements in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be multiplied.
694	**severe (694): Out of range: DIM argument to SUM has value 'dim' with ARRAY of rank 'rank'**
	FOR$IOS_F6739. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) of the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be summed.
695	**severe (695): Real zero raised to zero power**
	FOR$IOS_F6740. Zero of any type (real, complex, or integer) cannot be raised to zero power.
696	**severe (696): Real zero raised to negative power**
	FOR$IOS_F6741. Zero of any type (real, complex, or integer) cannot be raised to a negative power. Raising to a negative power inverts the operand.
697	**severe (697): Out of range: DIM argument to SUM has value 'dim'**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6742. The optional argument DIM specifies the dimension along which to sum the elements of an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be summed.
698	**severe (698): DIM argument ('dim') to EOSHIFT is out of range (1:'n')**
	FOR$IOS_F6743. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be shifted.
699	**severe (699): Shape mismatch (dimension 'dim') between ARRAY and BOUNDARY in EOSHIFT**
	FOR$IOS_F6744. The BOUNDARY argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the BOUNDARY must conform to the shape of the array being shifted in every dimension except the one being shifted along.
700	**severe (700): DIM argument to EOSHIFT is out of range ('dim')**
	FOR$IOS_F6745. The optional argument DIM specifies the dimension along which to perform an end-off shift in an array, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array holding the elements to be shifted.
701	**severe (701): Shape mismatch (dimension 'dim') between ARRAY and SHIFT in EOSHIFT**
	FOR$IOS_F6746. The SHIFT argument to EOSHIFT must be either scalar or an array one dimension smaller than the shifted array. If an array, the shape of the SHIFT must conform to the shape of the array being shifted in every dimension except the one being shifted along.
702	**severe (702): BOUNDARY argument to EOSHIFT has wrong LEN ('len1 instead of len2')**
	FOR$IOS_F6747. The character length of elements in the BOUNDARY argument and in the array being end-off shifted must be the same.
703	**severe (703): BOUNDARY has LEN 'len' instead of 'len' to EOSHIFT**
	FOR$IOS_F6748.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
704	**severe (704): Internal error - bad arguments to EOSHIFT** FOR$IOS_F6749.
705	**severe (705): GETARG: value of argument 'num' is out of range** FOR$IOS_F6750. The value used for the number of the command-line argument to retrieve with GETARG must be 0 or a positive integer. If the number of the argument to be retrieved is greater than the actual number of arguments, blanks are returned, but no error occurs.
706	**severe (706): FLUSH: value of LUNIT 'num' is out of range** FOR$IOS_F6751. The unit number specifying which I/O unit to flush to its associated file must be an integer between 0 and 2**31-1, inclusive. If the unit number is valid, but the unit is not opened, error F6752 is generated.
707	**severe (707): FLUSH: Unit 'n' is not connected** FOR$IOS_F6752. The I/O unit specified to be flushed to its associated file is not connected to a file.
708	**severe (708): Invalid string length ('len') to ICHAR** FOR$IOS_F6753. The character argument to ICHAR must have length 1.
709	**severe (709): Invalid string length ('len') to IACHAR** FOR$IOS_F6754. The character argument to IACHAR must have length 1.
710	**severe (710): Integer zero raised to negative power** FOR$IOS_F6755. Zero of any type (integer, real, or complex) cannot be raised to a negative power. Raising to a negative power inverts the operand.
711	**severe (711): INTEGER zero raised to zero power** FOR$IOS_F6756. Zero of any type (integer, real, or complex) cannot be raised to zero power.
712	**severe (712): SIZE argument ('size') to ISHFTC intrinsic out of range** FOR$IOS_F6757. The argument SIZE must be positive and must not exceed the bit size of the integer being shifted. The bit size of this integer can be determined with the function BIT_SIZE.
713	**severe (713): SHIFT argument ('shift') to ISHFTC intrinsic out of range**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6758. The argument SHIFT to ISHFTC must be an integer whose absolute value is less than or equal to the number of bits being shifted: either all bits in the number being shifted or a subset specified by the optional argument SIZE.
714	**severe (714): Out of range: DIM argument to LBOUND has value 'dim'**
	FOR$IOS_F6759. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
715	**severe (715): Out of range: DIM argument ('dim') to LBOUND greater than ARRAY rank 'rank'**
	FOR$IOS_F6760. The optional argument DIM specifies the dimension whose lower bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.
716	**severe (716): Out of range: DIM argument to MAXVAL has value 'dim'**
	FOR$IOS_F6761. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
717	**severe (717): Out of range: DIM argument to MAXVAL has value 'dim' with ARRAY of rank 'rank'**
	FOR$IOS_F6762. The optional argument DIM specifies the dimension along which maximum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.
718	**severe (718): Cannot allocate temporary array -- out of memory**
	FOR$IOS_F6763. There is not enough memory space to hold a temporary array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
719	**severe (719): Attempt to DEALLOCATE part of a larger object**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	FOR$IOS_F6764. An attempt was made to DEALLOCATE a pointer to an array subsection or an element within a derived type. The whole data object must be deallocated; parts cannot be deallocated.
720	**severe (720): Pointer in DEALLOCATE is ASSOCIATED with an ALLOCATABLE array** FOR$IOS_F6765. Deallocating a pointer associated with an allocatable target is illegal. Instead, deallocate the target the pointer points to, which frees memory and disassociates the pointer.
721	**severe (721): Attempt to DEALLOCATE an object which was not allocated** FOR$IOS_F6766. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation. The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.
722	**severe (722): Cannot ALLOCATE scalar POINTER -- out of memory** FOR$IOS_F6767. There is not enough memory space to allocate the pointer. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
723	**severe (723): DEALLOCATE: object not allocated/associated** FOR$IOS_F6768. You cannot deallocate an array unless it has been previously allocated. You cannot deallocate a pointer whose target was not created by allocation, or a pointer that has undefined association status. The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated.
724	**severe (724): Cannot ALLOCATE POINTER array -- out of memory** FOR$IOS_F6769. There is not enough memory space to allocate the POINTER array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
725	**severe (725): DEALLOCATE: Array not allocated** FOR$IOS_F6770. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.
726	**severe (726): DEALLOCATE: Character array not allocated** FOR$IOS_F6771. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.
727	**severe (727): Cannot ALLOCATE allocatable array -- out of memory** FOR$IOS_F6772. There is not enough memory space to hold the array. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.

728	**severe (728): Cannot allocate automatic object -- out of memory**
	FOR$IOS_F6773. There is not enough memory space to hold the automatic data object.
	Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Windows Control Panel or redefine the swap file size. Allocated arrays that are no longer needed should be deallocated.
	An automatic data object is an object that is declared in a procedure subprogram or interface, is not a dummy argument, and depends on a nonconstant expression. For example:
	`SUBROUTINE  EXAMPLE (N)`
	`  DIMENSION A (N, 5), B(10*N)`
	The arrays A and B in the example are automatic data objects.

729	**severe (729): DEALLOCATE failure: ALLOCATABLE array is not ALLOCATED** FOR$IOS_F6774. It is illegal to DEALLOCATE an array that is not allocated. You can check the allocation status of an array before deallocating with the ALLOCATED function.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
730	**severe (730): Out of range: DIM argument to MINVAL has value 'dim'** FOR$IOS_F6775. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
731	**severe (731): Out of range: DIM argument to MINVAL has value 'dim' with ARRAY of rank 'rank'** FOR$IOS_F6776. The optional argument DIM specifies the dimension along which minimum values are returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n* is the number of dimensions in array.
732	**severe (732): P argument to MOD is double precision zero** FOR$IOS_F6777. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.
733	**severe (733): P argument to MOD is integer zero** FOR$IOS_F6778. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.
734	**severe (734): P argument to MOD is real zero** FOR$IOS_F6779. MOD(A,P) is computed as A - INT(A/P) * P. So, P cannot be zero.
735	**severe (735): P argument to MODULO is real zero** FOR$IOS_F6780. >MODULO(A,P) for real numbers is computed as A - FLOOR(A/P) * P. So, P cannot be zero.
736	**severe (736): P argument to MODULO is zero** FOR$IOS_F6781. In the function, MODULO(A,P), P cannot be zero.
737	**severe (737): Argument S to NEAREST is zero** FOR$IOS_F6782. The sign of the S argument to NEAREST(X,S) determines the direction of the search for the nearest number to X, and cannot be zero.
738	**severe (738): Heap storage exhausted** FOR$IOS_F6783.
739	**severe (739): PUT argument to RANDOM_SEED is too small**

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation

FOR$IOS_F6784. The integer array PUT must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )           ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)              ! allocate array
CALL RANDOM_SEED(PUT = SEED)  ! set the seed
```

Note that RANDOM_SEED can be called with at most one argument at a time.

740 **severe (740): GET argument to RANDOM_SEED is too small**

FOR$IOS_F6785. The integer array GET must be greater than or equal to the number of integers the processor uses to set the seed value. This number can be determined by calling RANDOM_SEED with the SIZE argument. For example:

```
INTEGER, ALLOCATABLE SEED
CALL RANDOM_SEED( )           ! initialize processor
CALL RANDOM_SEED(SIZE = K)    ! get size of seed
ALLOCATE SEED(K)              ! allocate array
CALL RANDOM_SEED(GET = SEED)  ! get the seed
```

Note that RANDOM_SEED can be called with at most one argument at a time.

741 **severe (741): Recursive I/O reference**

FOR$IOS_F6786.

742 **severe (742): Argument to SHAPE intrinsic is not PRESENT**

FOR$IOS_F6787.

743 **severe (743): Out of range: DIM argument to UBOUND had value 'dim'**

FOR$IOS_F6788. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

744 **severe (744): DIM argument ('dim') to UBOUND greater than ARRAY rank 'rank'**

FOR$IOS_F6789. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
745	**severe (745): Out of range: UBOUND of assumed-size array with DIM==rank ('rank')** FOR$IOS_F6790. The optional argument DIM specifies the dimension whose upper bound is to be returned. An assumed-size array is a dummy argument in a subroutine or function, and the upper bound of its last dimension is determined by the size of actual array passed to it. Assumed-size arrays have no determined shape, and you cannot use UBOUND to determine the extent of the last dimension. You can use UBOUND to determine the upper bound of one of the fixed dimensions, in which case you must pass the dimension number along with the array name.
746	**severe (746): Out of range: DIM argument ('dim') to UBOUND greater than ARRAY rank** FOR$IOS_F6791. The optional argument DIM specifies the dimension whose upper bound is to be returned, and must be greater than or equal to 1 and less than or equal to the number of dimensions (rank) in the array. That is, 1 <= DIM <= *n*, where *n*is the number of dimensions in array.
747	**severe (747): Shape mismatch: Dimension 'shape' extents are 'ext1' and 'ext2'** FOR$IOS_F6792.
748	**severe (748): Illegal POSITION value** FOR$IOS_F6793. An illegal value was used with the POSITION specifier. POSITION accepts the following values: • 'ASIS' (the default) • 'REWIND' - on Fortran I/O systems, this is the same as 'ASIS' • 'APPEND'
749	**severe (749): Illegal ACTION value** FOR$IOS_F6794. An illegal value was used with the ACTION specifier. ACTION accepts the following values: • 'READ' • 'WRITE'

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
	• 'READWRITE' - the default
750	**severe (750): DELIM= specifier not allowed for an UNFORMATTED file** FOR$IOS_F6795. The DELIM specifier is only allowed for files connected for formatted data transfer. It is used to delimit character constants in list-directed an namelist output.
751	**severe (751): Illegal DELIM value** FOR$IOS_F6796. An illegal value was used with the DELIM specifier. DELIM accepts the following values: • 'APOSTROPHE' • 'QUOTE' • 'NONE' - the default
752	**severe (752): PAD= specifier not allowed for an UNFORMATTED file** FOR$IOS_F6797. The PAD specifier is only allowed for formatted input records. It indicates whether the formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.
753	**severe (753): Illegal PAD= value** FOR$IOS_F6798. An illegal value was used with the PAD specifier. PAD accepts the following values: • 'NO' • 'YES' - the default
754	**severe (754): Illegal CARRIAGECONTROL=value** FOR$IOS_F6799. An illegal value was used with the CARRIAGECONTROL specifier. CARRIAGECONTROL accepts the following values: • 'FORTRAN' - default if the unit is connected to a terminal or console • 'LIST' - default for formatted files • 'NONE' - default for unformatted files

Number	Severity Level, Number, and Message Text; Condition Symbol and Explanation
755	**severe (755): SIZE= specifier only allowed with ADVANCE='NO'** FOR$IOS_F6800. The SIZE specifier can only appear in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (indicating nonadvancing input).
756	**severe (756): Illegal character in binary input** FOR$IOS_F6801.
757	**severe (757): Illegal character in octal input** FOR$IOS_F6802.
758	**severe (758): End of record encountered** FOR$IOS_F6803.
759	**severe (759): Illegal subscript in namelist input record** FOR$IOS_F6804.

Footnotes:

[1] Identifies errors not returned by IOSTAT.

Signal Handling (Linux* OS and Mac OS* X only)

A `signal` is an ab normal event generated by one of various sources, such as:

- A user of a terminal
- Program or hardware error
- Request of another program
- When a process is stopped to allow access to the control terminal

You can optionally set certain events to issue signals, for example:

- When a process resumes after being stopped
- When the status of a child process changes
- When input is ready at the terminal

Some signals terminate the receiving process if no action is taken (optionally creating a `core` file), while others are simply ignored unless the process has requested otherwise.

Except for certain signals, calling the `signal` or `sigaction` routine allows specified signals to be ignored or causes an interrupt (transfer of control) to the location of a user-written signal handler.

You can establish one of the following actions for a signal with a call to `signal`:

- Ignore the specified signal (identified by number).
- Use the default action for the specified signal, which can reset a previously established action.
- Transfer control from the specified signal to a procedure to receive the signal, specified by name.

Calling the `signal` routine lets you change the action for a signal, such as intercepting an operating system signal and preventing the process from being stopped.

The table below shows the signals that the Intel Fortran RTL arranges to catch when a program is started:

Signal	Intel Fortran RTL message
SIGFPE	`Floating-point exception` (number 75)
SIGINT	`Process interrupted` (number 69)
SIGIOT	`IOT trap signal` (number 76)
SIGQUIT	`Process quit` (number 79)
SIGSEGV	`Segmentation fault` (number 174)
SIGTERM	`Process killed` (number 78)

Calling the `signal` routine (specifying the numbers for these signals) results in overwriting the signal-handling facility set up by the Intel Fortran RTL. The only way to restore the default action is to save the returned value from the first call to `signal`.

When using a debugger, it may be necessary to enter a command to allow the Intel Fortran RTL to receive and handle the appropriate signals.

Overriding the Default Run-Time Library Exception Handler

To override the default run-time library exception handler on Linux OS and Mac OS X, your application must call `signal` to change the action for the signal of interest.

For example, assume that you want to change the signal action to cause your application to call abort() and generate a `core` file.

The following example adds a function named clear_signal_ to call signal() and change the action for the SIGABRT signal:

```
#include <signal.h>
void clear_signal_()
{
signal (SIGABRT, SIG_DFL);
}
int myabort_()
{
abort();
return 0;
}
```

A call to the `clear_signal()` local routine must be added to main. Make sure that the call appears before any call to the local `myabort()` routine:

```
program aborts
integer i

call clear_signal()

i = 3
if (i < 5) then
call myabort()
end if
end
```

Using Traceback Information

Using Traceback Information Overview

When a Fortran program terminates due to a severe error condition, the Fortran run-time system displays additional diagnostic information after the run-time message.

The Fortran run-time system attempts to walk back up the call chain and produce a report of the calling sequence leading to the error as part of the default diagnostic message report. This is known as traceback. The minimum information displayed includes:

- The standard Fortran run-time error message text that explains the error condition.
- A tabular report that contains one line per call stack frame. This information includes at least the image name and a hexadecimal PC in that image.

The information displayed under the Routine, Line, and Source columns depends on whether your program was compiled with the `-traceback` (Linux* OS and Mac OS* X) or `/traceback` (Windows* OS) option.

For example, if `-traceback` or `/traceback` is specified, the displayed information might resemble the following:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image              PC        Routine         Line     Source
libifcorert.dll    1000A3B2  Unknown         Unknown  Unknown
libifcorert.dll    1000A184  Unknown         Unknown  Unknown
libifcorert.dll    10009324  Unknown         Unknown  Unknown
libifcorert.dll    10009596  Unknown         Unknown  Unknown
libifcorert.dll    10024193  Unknown         Unknown  Unknown
teof.exe           004011A9  AGAIN                21  teof.for
teof.exe           004010DD  GO                   15  teof.for
teof.exe           004010A7  WE                   11  teof.for
teof.exe           00401071  HERE                  7  teof.for
teof.exe           00401035  TEOF                  3  teof.for
teof.exe           004013D9  Unknown         Unknown  Unknown
teof.exe           004012DF  Unknown         Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown         Unknown  Unknown
```

If the same program is *not* compiled with the `-traceback` or `/traceback` option:

- The Routine name, Line number, and Source file columns would be reported as "Unknown."
- A link map file is usually needed to locate the cause of the error.

The `-traceback` or `/traceback` option provides program counter (PC) to source file line correlation information to appear in the displayed error message information, which simplifies the task of locating the cause of severe run-time errors.

For Fortran objects generated with `-traceback` or `/traceback`, the compiler generates additional information used by the Fortran run-time system to automatically correlate PC values to the routine name in which they occur, Fortran source file, and line number in the source file. This information is displayed in the run-time error diagnostic report.

Automatic PC correlation is only supported for Fortran code. For non-Fortran code, only the hexadecimal PC locations are reported.

Tradeoffs and Restrictions in Using Traceback

This topic describes tradeoffs and restrictions that apply to using traceback.

Effect on Image Size

Using the `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS) option to get automatic PC correlation increases the size of an image. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PCs with a map file is acceptable.

The approach of providing automatic correlation information in the image was used so that no run-time penalty is incurred by building the information "on the fly" as your application executes. No run-time diagnostic code is invoked unless your application is terminating due to a severe error.

C Compiler Omit Frame Pointer Option on Systems Using IA-32 Architecture

The following routines are used to walk the stack:

- For Windows OS, the Windows API routine StackWalk( ) in imagehlp.dll
- For Linux OS and Mac OS X, _Unwind_ForcedUnwind( ), _Unwind_GetIP( ), _Unwind_GetRegionStart( ) and _Unwind_GetGr( ) routines in libunwind.so

In an environment using IA-32 architecture, there are no firm software calling standards documented. Compiler developers are under no constraints to use machine registers in any particular way or to hook up procedures in any particular way. The stack walking routines listed above use a set of heuristics to determine how to walk the call stack. That is, they make a "best guess" to determine how a program reached a particular point in the call chain. With C code that has been compiled with Visual C++* using the Omit Frame Pointer option -- either `-fomit-frame-pointer` (Linux OS and Mac OS X) or `/Oy` (Windows OS) -- this "best guess" is not usually the correct one.

If you are mixing Fortran and C code and you are concerned about stack tracing, consider disabling the `-fomit-frame-pointer` or `/Oy` option in your C compilations. Otherwise, traceback will most likely not work for you.

Stack Trace Failure

Programs can fail for a number of reasons, often with unpredictable consequences. Memory corruption by erroneously executing code is one possibility. Stack memory can be corrupted in such a way that the attempt to trace the call stack will result in access violations or other undesirable consequences. The stack-tracing run-time code is guarded with a local exception filter. If the traceback attempt fails due to a hard detectable condition, the run-time will report this in its diagnostic output message as:

```
Stack trace terminated abnormally
```

Be forewarned, however: It is also possible for memory to be corrupted in such a way that a stack trace can seem to complete successfully with no hint of a problem. The bit patterns it finds in corrupted memory where the stack used to be, and then uses to access memory, may constitute perfectly valid memory addresses for the program to be accessing. They just do not

happen to have any connection to what the stack used to look like. So, if it appears that the stack walk completed normally, but the reported PCs make no sense to you, then consider ignoring the stack trace output in diagnosing your problem.

Another condition that will disable the stack trace process is your program exiting because it has exhausted virtual memory resources.

The stack trace can fail if the run-time system cannot dynamically load libunwind.so (Linux OS and Mac OS X) or imagehlp.dll (Windows OS) or cannot find the necessary routines from that library. In this case, you still get the basic run-time diagnostic message; you will not get any call stack information.

Linker /incremental:no Option on Windows Operating Systems

The following applies to Windows operating systems only.

When incremental linking is enabled, automatic PC correlation does not work. Use of incremental linking always disables automatic PC correlation even if you specify `/traceback` during compilation.

When you use incremental linking, the default hexadecimal (hex) PC values will still appear in the output. To correlate from the hexadecimal PC values to routine containing the PC addresses requires use of a linker map file. However, if you request a map file during linking, incremental linking becomes disabled. Thus to allow any PC values generated for a run-time problem to be helpful, incremental linking must be disabled.

In the integrated development environment, you can use the Call stack display, so incremental linking is not a problem.

Sample Programs and Traceback Information

The following sections provide sample programs that show the use of traceback to locate the cause of the error:

- Example: End-of-File Condition, Program teof
- Example: Machine Exception Condition, Program ovf
- Example: Using Traceback in Mixed Fortran/C Applications, Program FPING and CPONG

Note that the hex PC's and contents of registers displayed in these program outputs are meant as representative examples of typical output. The PC's will change over time, as the libraries and other tools used to create an image change.

Example: End-of-File Condition, Program teof

In the following example, a READ statement creates an End-Of-File error, which the application has not handled:

```
  program teof
  integer*4 i,res
  i=here( )
  end
 integer*4 function here( )
  here = we( )
  end
 integer*4 function we( )
  we = go( )
  end
 integer*4 function go( )
  go = again( )
  end
 integer*4 function again( )
  integer*4 a
  open(10,file='xxx.dat',form='unformatted',status='unknown')
  read(10) a
  again=a
  end
```

The diagnostic output that results when this program is built with traceback enabled and linked against the single-threaded, shared Fortran run-time library on the IA-32 architecture platform is similar to the following:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image            PC        Routine         Line     Source
libifcorert.dll  1000A3B2  Unknown         Unknown  Unknown
libifcorert.dll  1000A184  Unknown         Unknown  Unknown
libifcorert.dll  10009324  Unknown         Unknown  Unknown
libifcorert.dll  10009596  Unknown         Unknown  Unknown
libifcorert.dll  10024193  Unknown         Unknown  Unknown
teof.exe         004011A9  AGAIN                21  teof.for
teof.exe         004010DD  GO                   15  teof.for
teof.exe         004010A7  WE                   11  teof.for
teof.exe         00401071  HERE                  7  teof.for
teof.exe         00401035  TEOF                  3  teof.for
teof.exe         004013D9  Unknown         Unknown  Unknown
teof.exe         004012DF  Unknown         Unknown  Unknown
KERNEL32.dll     77F1B304  Unknown         Unknown  Unknown
```

The first line of the output is the standard Fortran run-time error message. What follows is the result of walking the call stack in reverse order to determine where the error originated. Each line of output represents a call frame on the stack. Since the application was compiled with

`-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS), the PCs that fall in Fortran code are correlated to their matching routine name, line number and source module. PCs that are not in Fortran code are not correlated and are reported as "Unknown."

The first five frames show the calls to routines in the Fortran run-time library (in reverse order). Since the application was linked against the single threaded, shared version of the library, the image name reported is either libifcore.so (Linux OS and Mac OS X) or libifcorert.dll (Windows OS). These are the run-time routines that were called to do the `READ` and upon detection of the EOF condition, were invoked to report the error. In the case of an unhandled I/O programming error, there will always be a few frames on the call stack down in run-time code like this.

The stack frame of real interest to the Fortran developer is the first frame in image teof.exe which shows that the error originated in the routine named AGAIN in source module `teof.for` at line 21. Looking in the source code at line 21, you can see the Fortran `READ` statement that incurred the end-of-file condition.

The next four frames show the trail of calls in the Fortran user code that led to the routine that got the error (TEOF->HERE->WE->GO->AGAIN).

Finally, the bottom three frames are routines which handled the startup and initialization of the program.

If this program had been linked against the single-threaded, static Fortran run-time library, the output would then look like:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image          PC        Routine        Line     Source
teof.exe       004067D2  Unknown        Unknown  Unknown
teof.exe       0040659F  Unknown        Unknown  Unknown
teof.exe       00405754  Unknown        Unknown  Unknown
teof.exe       004059C5  Unknown        Unknown  Unknown
teof.exe       00403543  Unknown        Unknown  Unknown
teof.exe       004011A9  AGAIN               21  teof.for
teof.exe       004010DD  GO                  15  teof.for
teof.exe       004010A7  WE                  11  teof.for
teof.exe       00401071  HERE                 7  teof.for
teof.exe       00401035  TEOF                 3  teof.for
teof.exe       004202F9  Unknown        Unknown  Unknown
teof.exe       00416063  Unknown        Unknown  Unknown
KERNEL32.dll   77F1B304  Unknown        Unknown  Unknown
```

Notice that the initial five stack frames now show routines in image `teof.exe`, not libifcore.so (Linux OS and Mac OS X) or `libifcorert.dll` (Windows OS). The routines are the same five run-time routines as previously reported for the shared library case but since the application was linked against the archive library libifcore.a (Linux OS and Mac OS X) or the static Fortran

run-time library `libifcore.lib` (Windows OS), the object modules containing these routines were linked into the application image (`teof.exe`). You can use a map file to determine the locations of uncorrelated PCs.

Now suppose the application was compiled *`without`* traceback enabled and, once again, linked against the single-threaded, static Fortran library. The diagnostic output would appear as follows:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
Image              PC        Routine       Line     Source
teof.exe           00406792  Unknown       Unknown  Unknown
teof.exe           0040655F  Unknown       Unknown  Unknown
teof.exe           00405714  Unknown       Unknown  Unknown
teof.exe           00405985  Unknown       Unknown  Unknown
teof.exe           00403503  Unknown       Unknown  Unknown
teof.exe           00401169  Unknown       Unknown  Unknown
teof.exe           004010A8  Unknown       Unknown  Unknown
teof.exe           00401078  Unknown       Unknown  Unknown
teof.exe           00401048  Unknown       Unknown  Unknown
teof.exe           0040102F  Unknown       Unknown  Unknown
teof.exe           004202B9  Unknown       Unknown  Unknown
teof.exe           00416023  Unknown       Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown       Unknown  Unknown
```

Without the correlation information in the image that `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS) previously supplied, the Fortran run-time system cannot correlate PC's to routine name, line number, and source file. You can still use the map file to at least determine the routine names and what modules they are in.

Remember that compiling with `-traceback` or `/traceback` increases the size of your application's image because of the extra PC correlation information included in the image. You can see if the extra traceback information is included in an image (checking for the presence of a `.trace` section) by entering:

```
objdump -h your_app.exe (Linux OS)
otool -l your_app.exe (Mac OS X)
link -dump -summary your_app.exe (Windows OS)
```

Build your application with and without traceback and compare the file size of each image. Check the file size with a simple directory command.

For this simple `teof.exe` example, the traceback correlation information adds about 512 bytes to the image size. In a real application, this would probably be much larger. For any application, the developer must decide if the increase in image size is worth the benefit of automatic PC correlation or if manually correlating PC's with a map file is acceptable.

If an error occurs when traceback was requested during compilation, the run-time library will produce the correlated call stack display.

If an error occurs when traceback was disabled during compilation, the run-time library will produce the uncorrelated call stack display.

If you do not want to see the call stack information displayed, you can set the environment variable `FOR_DISABLE_STACK_TRACE` to true. You will still get the Fortran run-time error message:

```
forrtl: severe (24): end-of-file during read, unit 10, file E:\USERS\xxx.dat
```

Example: Machine Exception Condition, Program ovf

The following program generates a floating-point overflow exception when compiled with `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS):

```
  program ovf
  real*4 a
  a=1e37
  do i=1,10
    a=hey(a)
  end do
  print *, 'a= ', a
  end
 real*4 function hey(b)
  real*4 b
  hey = watch(b)
  end
 real*4 function watch(b)
  real*4 b
  watch = out(b)
  end
 real*4 function out(b)
  real*4 b
  out = below(b)
  end
 real*4 function below(b)
  real*4 b
  below = b*10.0e0
  end
```

Assume this program is compiled with the following:

- `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS)
- `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)
- `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS)

On a system based on IA-32 architecture, the traceback output is similar to the following:

```
forrtl: error (72): floating overflow
Image              PC        Routine          Line     Source
ovf.exe            00401161  BELOW               29  ovf.f90
ovf.exe            0040113C  OUT                 24  ovf.f90
ovf.exe            0040111B  WATCH               19  ovf.f90
ovf.exe            004010FA  HEY                 14  ovf.f90
ovf.exe            0040105B  OVF                  7  ovf.f90
ovf.exe            00432429  Unknown         Unknown  Unknown
ovf.exe            00426C74  Unknown         Unknown  Unknown
KERNEL32.dll       77F1B9EA  Unknown         Unknown  Unknown
```

Notice that unlike the previous example of an unhandled I/O programming error, the stack walk can begin right at the point of the exception. There are no run-time routines on the call stack to dig through. The overflow occurs in routine BELOW at PC 00401161, which is correlated to line 29 of the source file `ovf.f90`.

When the program is compiled at a higher optimization level of `O2`, along with `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows) and `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS), the traceback output appears as follows:

```
forrtl: error (72): floating overflow
Image              PC        Routine        Line       Source
ovf.exe            00401070  OVF                29    ovf.f90
ovf.exe            004323E9  Unknown        Unknown    Unknown
ovf.exe            00426C34  Unknown        Unknown    Unknown
KERNEL32.dll       77F1B9EA  Unknown        Unknown    Unknown
```

With `/O2`, the entire program has been inlined.

The main program, `OVF`, no longer calls routine `HEY`. While the output is not quite what one might have expected intuitively, it is still entirely correct. You need to keep in mind the effects of compiler optimization when you interpret the diagnostic information reported for a failure in a release image.

If the same image were executed again, this time with the environment variable called `TBK_ENABLE_VERBOSE_STACK_TRACE` set to True, you would also see a dump of the exception context record at the time of the error. Here is an excerpt of how that might appear on a system using IA-32 architecture:

```
forrtl: error (72): floating overflow
Hex Dump Of Exception Record Context Information:
Exception Context:  Processor Control and Status Registers.
EFlags:  00010212
CS:  0000001B  EIP:  00401161  SS:   00000023  ESP:  0012FE38  EBP:  0012FE60
Exception Context:  Processor Integer Registers.
EAX: 00444488  EBX:  00000009  ECX:  00444488  EDX:  00000002
ESI: 0012FBBC  EDI:  F9A70030
```

```
Exception Context:  Processor Segment Registers.
DS:  00000023  ES:   00000023  FS:   00000038  GS:   00000000
Exception Context:  Floating Point Control and Status Registers.
ControlWord:  FFFF0262  ErrorOffset:    0040115E  DataOffset:    0012FE5C
StatusWord:   FFFFF8A8  ErrorSelector:  015D001B  DataSelector:  FFFF0023
TagWord:      FFFF3FFF  Cr0NpxState:    00000000
Exception Context:  Floating Point RegisterArea.
RegisterArea[00]:  4080BC143F4000000000  RegisterArea[10]:  F7A0FFFFFFFF77F9D860
RegisterArea[20]:  00131EF0000800060012  RegisterArea[30]:  00000012F7C002080006
RegisterArea[40]:  02080006000000000000  RegisterArea[50]:  0000000000000012F7D0
RegisterArea[60]:  00000000000000300000  RegisterArea[70]:  FBBC000000300137D9EF
...
```

Example: Using Traceback in Mixed Fortran/C Applications, Program FPING and CPONG

Consider the following example that shows how the traceback output might appear in a mixed Fortran/C application. The main program is a Fortran program named `FPING`. Program `FPING` triggers a chain of function calls which are alternately Fortran and C code. Eventually, the C routine named Unlucky is called, which produces a floating divide-by-zero error.

Source module `FPING.FOR` contains the Fortran function definitions, each of which calls a C routine from source module `CPONG.C`. The program `FPING.FOR` is compiled with the following options:

- `-fpe 0` (Linux OS and Mac OS X) or `/fpe:0` (Windows OS)
- `-traceback` (Linux OS and Mac OS X) or `/traceback` (Windows OS)
- `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS)

On the IA-32 architecture platform. the program traceback output resembles the following:

```
forrtl: error (73): floating divide by zero
Image              PC        Routine       Line     Source
fping.exe          00401161  Unknown       Unknown  Unknown
fping.exe          004010DC  DOWN4              58  fping.for
fping.exe          0040118F  Unknown       Unknown  Unknown
fping.exe          004010B6  DOWN3              44  fping.for
fping.exe          00401181  Unknown       Unknown  Unknown
fping.exe          00401094  DOWN2              31  fping.for
fping.exe          00401173  Unknown       Unknown  Unknown
fping.exe          00401072  DOWN1              18  fping.for
fping.exe          0040104B  FPING               5  fping.for
fping.exe          004013B9  Unknown       Unknown  Unknown
fping.exe          004012AF  Unknown       Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown       Unknown  Unknown
```

Notice that the stack frames contributed by Fortran routines can be correlated to a routine name, line number, and source module but those frames contributed by C routines cannot be correlated. Remember, even though the stack can be walked in reverse, and PCs reported, the information necessary to correlate the PC to a routine name, line number, and so on, is contributed to the image from the objects generated by the Fortran compiler. The C compiler does not have this capability. Also remember that you only get the correlation information if you specify the `-traceback` or `/traceback` option for your Fortran compiles.

The top stack frame cannot be correlated to a routine name because it is in C code. You can examine the map file for the application; if you do so, you will see that the reported PC, 00401161, is greater than the start of routine _Unlucky, but less than the start of routine _down1_C. This means that the error occurred in routine _Unlucky.

In a similar manner, the other PCs reported as "Unknown" can be correlated to a routine name using the map file.

When examining traceback output (or any type of diagnostic output, for that matter), it is important to keep in mind the effects of compiler optimization. The Fortran source module in the above example was built with optimization turned off. Look at the output when optimizations are enabled with `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS):

```
forrtl: error (73): floating divide by zero
Image              PC        Routine     Line     Source
fping.exe          00401111  Unknown     Unknown  Unknown
fping.exe          0040109D  DOWN4            58  fping.for
fping.exe          0040113F  Unknown     Unknown  Unknown
fping.exe          00401082  DOWN3            44  fping.for
fping.exe          00401131  Unknown     Unknown  Unknown
fping.exe          0040106B  DOWN2            31  fping.for
fping.exe          00401123  Unknown     Unknown  Unknown
fping.exe          00401032  FPING            18  fping.for
fping.exe          00401369  Unknown     Unknown  Unknown
fping.exe          0040125F  Unknown     Unknown  Unknown
KERNEL32.dll       77F1B304  Unknown     Unknown  Unknown
```

From the traceback output, it would appear that routine DOWN1 was never called. In fact, it has not been called. At the higher optimization level, the compiler has inlined function DOWN1 so that the call to routine down1_C is now made from FPING. The correlated line number still points to the correct line in the source code.

Finally, suppose the example Fortran code is redesigned with each of the Fortran routines split into separate source modules. Here is what the traceback output would look like with the redesigned code:

```
forrtl: error (73): floating divide by zero
Image              PC        Routine      Line     Source
fpingmain.exe      00401171  Unknown      Unknown  Unknown
```

```
fpingmain.exe    004010ED  DOWN4            12  fping4.for
fpingmain.exe    0040119F  Unknown      Unknown  Unknown
fpingmain.exe    004010C1  DOWN3            11  fping3.for
fpingmain.exe    00401191  Unknown      Unknown  Unknown
fpingmain.exe    00401099  DOWN2            11  fping2.for
fpingmain.exe    00401183  Unknown      Unknown  Unknown
fpingmain.exe    00401073  DOWN1            11  fping1.for
fpingmain.exe    0040104B  FPING             5  fpingmain.for
fpingmain.exe    004013C9  Unknown      Unknown  Unknown
fpingmain.exe    004012BF  Unknown      Unknown  Unknown
KERNEL32.dll     77F1B304  Unknown      Unknown  Unknown
```

Notice that the line number and source file correlation information has changed to reflect the new design of the code.

Here are the sources used in the above examples:

```
***********************************
FPING.FOR
***********************************
program fping
real*4 a,b
a=-10.0
b=down1(a)
end
real*4 function down1(b)
real*4 b
!DEC$ IF DEFINED(_X86_)
INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'_down1_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'down1_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down1_C
down1 = down1_C(b)
end
real*4 function down2(b)
real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'_down2_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'down2_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down2_C
down2 = down2_C(b)
end
real*4 function down3(b)
real*4 b [VALUE]
```

```
!DEC$ IF DEFINED(_X86_)
INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'_down3_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'down3_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down3_C
down3 = down3_C(b)
end
real*4 function down4(b)
real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'_Unlucky'] (a,c)
!DEC$ ELSE
INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'Unlucky'] (a,c)
!DEC$ ENDIF
REAL*4 a [VALUE]
REAL*4 c [REFERENCE]
END
real*4 a
call Unlucky(b,a)
down4 = a
end
***********************************
CPONG.C
***********************************
#include <math.h>
extern float __stdcall DOWN2 (float n);
extern float __stdcall DOWN3 (float n);
extern float __stdcall DOWN4 (float n);
int Fact( int n )
{
   if (n > 1)
       return( n * Fact( n - 1 ));
   return 1;
}
void Pythagoras( float a, float b, float *c)
{
   *c = sqrt( a * a + b * b );
}
void Unlucky( float a, float *c)
{
float b=0.0;
   *c = a/b;
}
float down1_C( float a )
{
   return( DOWN2( a ));
}
float down2_C( float a )
```

```
{
   return( DOWN3( a ));
}
float down3_C( float a )
{
   return( DOWN4( a ));
}
************************************
FPINGMAIN.FOR
************************************
program fping
real*4 a,b
a=-10.0
b=down1(a)
end
************************************
FPING1.FOR
************************************
real*4 function down1(b)
real*4 b
!DEC$ IF DEFINED(_X86_)
INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'_down1_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down1_C [C,ALIAS:'down1_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down1_C
down1 = down1_C(b)
end
************************************
FPING2.FOR
************************************
real*4 function down2(b)
real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'_down2_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down2_C [C,ALIAS:'down2_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down2_C
down2 = down2_C(b)
end
************************************
FPING3.FOR
************************************
real*4 function down3(b)
real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
```

```
INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'_down3_C'] (n)
!DEC$ ELSE
INTERFACE TO REAL*4 FUNCTION down3_C [C,ALIAS:'down3_C'] (n)
!DEC$ ENDIF
REAL*4 n [VALUE]
END
real*4 down3_C
down3 = down3_C(b)
end
**********************************
FPING4.FOR
**********************************
real*4 function down4(b)
real*4 b [VALUE]
!DEC$ IF DEFINED(_X86_)
INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'_Unlucky'] (a,c)
!DEC$ ELSE
INTERFACE TO SUBROUTINE Unlucky [C,ALIAS:'Unlucky'] (a,c)
!DEC$ ENDIF
REAL*4 a [VALUE]
REAL*4 c [REFERENCE]
END
real*4 a
call Unlucky(b,a)
down4 = a
end
```

Obtaining Traceback Information with TRACEBACKQQ

You can obtain traceback information in your application by calling the TRACEBACKQQ routine.

`TRACEBACKQQ` allows an application to initiate a stack trace. You can use this routine to report application detected errors, use it for debugging, and so on. It uses the standard stack trace support in the Intel® Fortran run-time system to produce the same output that the run-time system produces for unhandled errors and exceptions (severe error message). The `TRACEBACKQQ` subroutine generates a stack trace showing the program call stack as it was leading up to the point of the call to `TRACEBACKQQ`.

The error message string normally included from the run-time support is replaced with the user-supplied message text or omitted if no user string is specified. Traceback output is directed to the target destination appropriate for the application type, just as it is when traceback is initiated internally by the run-time support.

In the most simple case, a user can generate a stack trace by coding the call to TRACEBACKQQ with no arguments:

```
CALL TRACEBACKQQ()
```

This call causes the run-time library to generate a traceback report with no leading header message, from wherever the call site is, and terminate execution.

You can specify arguments that generate a stack trace with the user-supplied string as the header and instead of terminating execution, return control to the caller to continue execution of the application. For example:

```
CALL TRACEBACKQQ(STRING="Done with pass 1",USER_EXIT_CODE=-1)
```

By specifying a user exit code of -1, control returns to the calling program. Specifying a user exit code with a positive value requests that specified value be returned to the operating system. The default value is 0, which causes the application to abort execution.

Portability Considerations 16

Portability Considerations Overview

This section presents topics to help you understand how language standards, operating system differences, and computing hardware influence your use of Intel® Fortran and the portability of your programs.

Your program is portable if you can implement it on one hardware-software platform and then move it to additional systems with a minimum of changes to the source code. Correct results on the first system should be correct on the additional systems. The number of changes you might have to make when moving your program varies significantly. You might have no changes at all (strictly portable), or so many (non-portable customization) that it is more efficient to design or implement a new program. Most programs in their lifetime will need to be ported from one system to another, and this section can help you write code that makes this easy.

See also:

- Understanding Fortran Language Standards and related topics
- Minimizing Operating System-Specific Information
- Storing and Representing Data
- Formatting Data for Transportability
- Portability Library Overview

Understanding Fortran Language Standards

Understanding Fortran Language Standards Overview

A language standard specifies the form and establishes the interpretation of programs expressed in the language. Its primary purpose is to promote, among vendors and users, portability of programs across a variety of systems.

The vendor-user community has adopted four major Fortran language standards. ANSI (American National Standards Institute) and ISO (International Standards Organization) are the primary organizations that develop and publish the standards.

The major Fortran language standards are:

- FORTRAN IV

American National Standard Programming Language FORTRAN, ANSI X3.9-1966. This was the first attempt to standardize the languages called FORTRAN by many vendors.

- FORTRAN 77

  American National Standard Programming Language FORTRAN, ANSI X3.9-1978. This standard added new features based on vendor extensions to FORTRAN IV and addressed problems associated with large-scale projects, such as improved control structures.

- Fortran 90

  American National Standard Programming Language Fortran, ANSI X3.198-1992 and International Standards Organization, ISO/IEC 1539: 1991, Information technology -- Programming languages -- Fortran. This standard emphasizes modernization of the language by introducing new developments. For information about differences between Fortran 90 and FORTRAN 77, see the *Fortran Language Reference Manual*.

- Fortran 95

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1: 1997(E), Information technology -- Programming languages -- Fortran. This standard introduces certain language elements and corrections into Fortran 90. Fortran 95 includes Fortran 90 and most features of FORTRAN 77. For information about differences between Fortran 95 and Fortran 90, see the *Fortran Language Reference Manual*.

- Fortran 2003

  American National Standard Programming Language Fortran and International Standards Organization, ISO/IEC 1539-1:2004, Information technology -- Programming languages -- Fortran. This standard introduces extended support for exception handling, object-oriented programming, and improved interoperability with the C language. For more information on supported Fortran 2003 features, see the Fortran Language Reference Manual.

Although a language standard seeks to define the form and the interpretation uniquely, a standard may not cover all areas of interpretation. It may also include some ambiguities. You need to carefully craft your program in these cases so that you get the answers that you want when producing a portable program.

Using Standard Features and Extensions

Use standard features to achieve the greatest degree of portability for your Intel Fortran programs. You can design a robust implementation to improve the portability of your program, or you can choose to use extensions to the standard to increase the readability, functionality, and efficiency of your programs. You can ensure your program enforces the Fortran standard by using the `-stand` (Linux* OS and Mac OS* X) or `/stand` (Windows* OS) compiler option with the appropriate keyword (`f90`, `f95`, or `f03`) to flag extensions. The `none` keyword turns

off enforcement of a particular Fortran standard. You can also use the following compiler options to set the Fortran standard: `-std90` or `/std90`, `-std95` or `/std95`, and `-std03` or `/std03`. The default is std03, which diagnoses exceptions to the Fortran 2003 standard.

Not all Fortran standard extensions cause problems when porting to other platforms. Many extensions are supported on a wide range of platforms, and if a system you are porting a program to supports an extension, there is no reason to avoid using it. There is no guarantee, however, that the same feature on another system will be implemented in the same way as it is in Intel Fortran. Only the Fortran standard is guaranteed to coexist uniformly on all platforms.

Intel® Fortran supports many language extensions on multiple platforms, including Windows, Linux, and Mac OS X operating systems. The Intel® *Fortran Language Reference Manual* identifies whether each language element is supported on other platforms.

It is a good programming practice to declare any external procedures either in an EXTERNAL statement or in a procedure interface block, for the following reasons:

- The Fortran 90 standard added many new intrinsic procedures to the language. Programs that conformed to the FORTRAN 77 standard may include nonintrinsic functions or subroutines having the same name as new Fortran 90 procedures.
- Some processors include nonstandard intrinsic procedures that might conflict with procedure names in your program.

If you do not explicitly declare the external procedures and the name duplicates an intrinsic procedure, the processor calls the intrinsic procedure, not your external routine. For more information on how the Fortran compiler resolves name definitions, see Resolving Procedure References.

Using Compiler Optimizations

Many Fortran compilers perform code-generation optimizations to increase the speed of execution or to decrease the required amount of memory for the generated code. Although the behaviors of both the optimized and nonoptimized programs fall within the language standard specification, different behaviors can occur in areas not covered by the language standard. Compiler optimization especially can influence floating-point numeric results.

The Intel® Fortran compiler can perform optimizations to increase execution speed and to improve floating-point numerical consistency.

Floating-point consistency refers to obtaining results consistent with the IEEE binary floating-point standards. For more information, see the `-fltconsistency` (Linux OS and Mac OS X) or `/fltconsistency` (Windows OS) option.

Unless you properly design your code, you may encounter numerical difficulties when you optimize for fastest execution. The `-nofltconsistency` or `/nofltconsistency` option uses the floating-point registers, which have a higher precision than stored variables, whenever possible. This tends to produce results that are inconsistent with the precision of stored variables.

The `-fltconsistency` or `/fltconsistency` option can improve the consistency of generated code by rounding results of statement evaluations to the precision of the standard data types, but it does produce slower execution times.

See also Optimizing Applications.

Minimizing Operating System-Specific Information

The operating system influences your program both externally and internally. For increased portability, you need to minimize the amount of operating-system-specific information required by your program. The Fortran language standards do not specify this information.

Operating-system-specific information consists of nonintrinsic extensions to the language, compiler and linker options, and possibly the graphical user interface of Windows. Input and output operations use devices that may be system-specific, and may involve a file system with system-specific record and file structures.

The operating system also governs resource management and error handling. You can depend on default resource management and error handling mechanisms or provide mechanisms of your own. For information on special library routines to help port your program from one system to another, see Portability Library Overview and related topics.

The minimal interaction with the operating system is for input/output (I/O) operations and usually consists of knowing the standard units preconnected for input and output. You can use default file units with the asterisk (*) unit specifier.

To increase the portability of your programs across operating systems, consider the following:

- Do not assume the use of a particular type of file system.
- Do not embed filenames or paths in the body of your program. Define them as constants at the beginning of the program or read them from input data.
- Do not assume a particular type of standard I/O device or the "size" of that device (number of rows and columns).
- Do not assume display attributes for the standard I/O device. Some environments do not support attributes such as color, underlined text, blinking text, highlighted text, inverse text, protected text, or dim text.

Storing and Representing Data

The Fortran language standard specifies little about the storage of data types.

This loose specification of storage for data types results from a great diversity of computing hardware. This diversity poses problems in representing data and especially in transporting stored data among a multitude of systems. The size (as measured by the number of bits) of a

storage unit (a word, usually several bytes) varies from machine to machine. In addition, the ordering of bits within bytes and bytes within words varies from one machine to another. Furthermore, binary representations of negative integers and floating-point representations of real and complex numbers take several different forms.

If you are careful, you can avoid most of the problems involving data storage. The simplest and most reliable means of transferring data between dissimilar systems is in *character* and not binary form. Simple programming practices ensure that your data as well as your program is portable.

See also Supported Native and Nonnative Numeric Formats.

Formatting Data for Transportability

You can achieve the highest transportability of your data by formatting it as 8-bit character data. Use a standard character set such as the ASCII standard for encoding your character data. Although this practice is less efficient than using binary data, it will save you from shuffling and converting your data.

If you are transporting your data by means of a record-structured medium, it is best to use the Fortran sequential formatted (as character data) form. You can also use the direct formatted form, but you need to know the record length of your data.

Remember also that some systems use a carriage return/linefeed pair as an end-of-record indicator, while other systems use linefeed only. If you use either the direct unformatted or the sequential unformatted form, there might be system-dependent values embedded within your data that complicate its transport.

Implementing a strictly portable solution requires a careful effort. Maximizing portability may also mean making compromises to the efficiency and functionality of your solution. If portability is not your highest priority, you can use some of the techniques that appear in later sections to ease your task of customizing a solution.

See Also

- Portability Considerations
- Supported Native and Nonnative Numeric Formats
- Porting Nonnative Data
- Methods of Specifying the Data Format

Troubleshooting

17

Troubleshooting Your Application

The following lists some of the most basic problems you can encounter during application development and gives suggestions for troubleshooting:

- Source code does not compile correctly.

  If your source code fails to compile, check for unsupported language extensions. Typically, these produce a syntax error. The best way to resolve problems of this nature is to rewrite the source code so it conforms to the supported Fortran standards and does not contain unsupported extensions.

- Program does not run produce expected results.

  Use test scenarios that ensure the output matches your expectations. If a test fails, try compiling the files using the `-O0` (Linux* OS and Mac OS* X) or `/Od` (Windows* OS) option, which turns off the optimizer. If the test still fails, it is likely that the source code contains a problem. If your program runs successfully with `-O0` (Linux OS and Mac OS X) or `/Od` (Windows OS), but fails with the default `-O2` (Linux OS and Mac OS X) or `/O2` (Windows OS), you need to determine which file or files are causing the problem.

- Program runs slowly.

  Try to determine where your program spends most of its time. Such an analysis will show you which lines of your program are using the most execution time. See the Optimizing Applications book for additional guidelines that will help you optimize performance and gain speed.

Reference Information

18

Key Compiler Files Summary

The following table lists the key files that are installed for use by the compiler.

\bin Files	
File	**Description**
`codecov`	Executable for the Code-coverage tool
`fortcom`	Executable used by the compiler
`fpp`	Fortran preprocessor
`ias` (Linux* OS and Mac OS* X)	Assembler for systems using IA-64 architecture
`idis` (Windows* OS)	Disassembler for systems using IA-64 architecture
`ifortvars`	File to set environment variables
`ifort.cfg`	Configuration file for use from command line
`ifort`	Intel® Fortran Compiler
`ifortbin` (Linux OS and Mac OS X)	Executable used by the compiler
`map_opts`	Utility used for option translation
`profmerge`	Utility used for Profile Guided Optimizations
`proforder`	Utility used for Profile Guided Optimizations
`tselect`	Test prioritization tool
`uninstall.sh` (Linux OS and Mac OS X)	Uninstall utility
`xiar` (Linux OS)	Tool used for Interprocedural Optimizations

\bin Files	
File	**Description**
`xilibtool` (Mac OS X) `xilib` (Windows OS)	
`xild` (Linux OS and Mac OS X) `xilink` (Windows OS)	Tool used for Interprocedural Optimizations

For a list of the files installed in the `lib` directory, see Supplied Libraries.

Compiler Limits

The amount of data storage, the size of arrays, and the total size of executable programs are limited only by the amount of process virtual address space available, as determined by system parameters.

The table below shows the limits to the size and complexity of a single Intel® Fortran program unit and to individual statements contained within it:

Language Element	Limit
Actual number of arguments per CALL or function reference	Limited only by memory constraints
Arguments in a function reference in a specification expression	255
Array dimensions	7
Array elements per dimension	9,223,372,036,854,775,807 = 2**31-1 on systems using IA-32 architecture; 2**63-1 on systems using Intel® 64 and IA-64 architectures; plus limited by current memory configuration
Constants: character and Hollerith	7198
Constants: characters read in list-directed I/O	2048 characters

Language Element	Limit
Continuation lines - free form	Depends on line complexity and the number of lexical tokens allowed.
Continuation lines - fixed form	Depends on line complexity and the number of lexical tokens allowed.
Data and I/O implied DO nesting	7
DO and block IF statement nesting (combined)	256
DO loop index variable	9,223,372,036,854,775,807= 2**63-1
Format group nesting	8
Fortran statement length	2048 characters
Fortran source line length	fixed form: 72 (or 132 if `/extend_source` is in effect) characters; free form: 7200 characters
INCLUDE file nesting	20 levels
Labels in computed or assigned GOTO list	Limited only by memory constraints
Lexical tokens per statement	40000
Named common blocks	Limited only by memory constraints
Nesting of array constructor implied DOs	7
Nesting of input/output implied DOs	7
Nesting of interface blocks	Limited only by memory constraints
Nesting of DO, IF, or CASE constructs	Limited only by memory constraints
Nesting of parenthesized formats	Limited only by memory constraints
Number of arguments to MIN and MAX	Limited only by memory constraints
Number of digits in a numeric constant	Limited by statement length

Language Element	Limit
Parentheses nesting in expressions	Limited only by memory constraints
Structure nesting	30
Symbolic name length	63 characters
Width field for a numeric edit descriptor	2**31 - 1

See the product Release Notes for more information on memory limits for large data objects.

Part II

Compiler Options

Topics:

- Overview: Compiler Options
- Alphabetical Compiler Options
- Quick Reference Guides and Cross References
- Related Options

Overview: Compiler Options

19

This document provides details on all current Linux* OS, Mac OS* X, and Windows* OS compiler options.

It provides the following information:

- New options
  This topic lists new compiler options in this release.
- Deprecated
  This topic lists deprecated and removed compiler options for this release. Some deprecated options show suggested replacement options.
- Alphabetical Compiler Options
  This topic is the main source in the documentation set for general information on all compiler options. Options are described in alphabetical order. The Overview describes what information appears in each compiler option description.
- Quick Reference Guide and Cross Reference
  This topic contains tables summarizing compiler options. The tables show the option name, a short description of the option, the default setting for the option, and the equivalent option on the operating system, if any.
- Related Options
  This topic lists related options that can be used under certain conditions.

In this guide, compiler options are available on all supported operating systems and architectures unless otherwise identified.

For further information on compiler options, see Building Applications and Optimizing Applications.

Functional Groupings of Compiler Options

To see functional groupings of compiler options, specify a functional category for option `help` on the command line. For example, to see a list of options that affect diagnostic messages displayed by the compiler, enter one of the following commands:

```
-help diagnostics     ! Linux and Mac OS X systems

/help diagnostics     ! Windows systems
```

For details on the categories you can specify, see help.

New Options

This topic lists the options that provide new functionality in this release.

Some compiler options are only available on certain systems, as indicated by these labels:

Label	Meaning
i32	The option is available on systems using IA-32 architecture.
i64em	The option is available on systems using Intel® 64 architecture.
i64	The option is available on systems using IA-64 architecture.

If no label appears, the option is available on all supported systems.

If "only" appears in the label, the option is only available on the identified system.

For more details on the options, refer to the Alphabetical Compiler Options section.

For information on conventions used in this table, see Conventions.

New compiler options are listed in tables below:

- The first table lists new options that are available on Windows* systems.
- The second table lists new options that are available on Linux* and Mac OS* X systems. If an option is only available on one of these operating systems, it is labeled.

Windows* OS Options	Description	Default
`/arch:IA32` (i32 only)	Generates code that will run on any Pentium or later processor.	OFF
`/arch:SSE3` (i32, i64em)	Optimizes for Intel® Streaming SIMD Extensions 3 (Intel® SSE3).	OFF
`/arch:SSSE3` (i32, i64em)	Optimizes for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3).	OFF
`/arch:SSE4.1` (i32, i64em)	Optimizes for Intel® Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators.	OFF

Windows* OS Options	Description	Default
/assume:[no]ieee_fpe_flags	Determines whether the floating-point exception and status flags are saved on routine entry and restored on routine exit.	OFF
/assume:[no]old_logical_ldio	Determines whether NAMELIST and list-directed input accept logical values for numeric IO-list items.	ON
/assume:[no]old_maxminloc	Determines the results of the intrinsics MAXLOC and MINLOC when given an empty array as an argument.	ON
/debug:[no]parallel(i32, i64em)	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection of the Intel® Parallel Debugger Extension.	OFF
/fpe-all:n	Allows some control over floating-point exception handling for each routine in a program at run-time.	/fpe-all:3
/GS (i32, i64em)	Determines whether the compiler generates code that detects some buffer overruns.	/GS-
/homeparams	Tells the compiler to store parameters passed in registers to the stack.	OFF
/hotpatch (i32, i64em)	Tells the compiler to prepare a routine for hotpatching	OFF
/QaxSSE2 (i32, i64em)	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2.	OFF
/QaxSSE3 (i32, i64em)	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture.	OFF

Windows* OS Options	Description	Default
/QaxSSSE3 (i32, i64em)	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family.	OFF
/QaxSSE4.1 (i32, i64em)	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.	OFF
/QaxSSE4.2 (i32, i64em)	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.	OFF
/Qdiag-enable:sc-parallel (i32, i64em)	Enables analysis of parallelization in source code (parallel lint diagnostics).	OFF
/Qdiag-error-limit:n	Specifies the maximum number of errors allowed before compilation stops.	n=30
/Qdiag-once:id[,id,...]	Tells the compiler to issue one or more diagnostic messages only once.	OFF
/Qdiag-error-limit:n	Specifies the maximum number of errors allowed before compilation stops.	n=30
/Qdiag-once:id[,id,...]	Tells the compiler to issue one or more diagnostic messages only once.	OFF
/Qfast-transcendentals	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.	OFF

Windows* OS Options	Description	Default
/Qfma (i64 only)	Enables the combining of floating-point multiplies and add/subtract operations.	ON
/Qfp-relaxed (i64 only)	Enables use of faster but slightly less accurate code sequences for math functions.	OFF
/Qinstruction:[no]movbe (i32, i64em)	Determines whether MOVBE instructions are generated for Intel processors.	OFF
/Qimsl	Tells the compiler to link to the IMSL* library.	OFF
/Qmkl	Tells the compiler to link to certain parts of the Intel® Math Kernel Library.	OFF
/Qopenmp-link:library	Controls whether the compiler links to static or dynamic OpenMP run-time libraries.	/Qopenmp-link:dynamic
/Qopenmp-threadprivate:type	Lets you specify an OpenMP* threadprivate implementation.	/Qopenmp-threadprivate:legacy
/Qopt-block-factor:n	Lets you specify a loop blocking factor.	OFF
/Qopt-jump-tables:keyword	Enables or disables generation of jump tables for switch statements.	/Qopt-jump-tables:default
/Qopt-loadpair (i64 only)	Enables loadpair optimization.	/Qopt-loadpair-
/Qopt-mod-versioning (i64 only)	Enables versioning of modulo operations for certain types of operands.	/Qopt-mod-versioning-

Windows* OS Options	Description	Default
/Qopt-prefetch-initial-values (i64 only)	Enables or disables prefetches that are issued before a loop is entered.	/Qopt-prefetch-initial-values
/Qopt-prefetch-issue-excl-hint (i64 only)	Determines whether the compiler issues prefetches for stores with exclusive hint.	/Qopt-prefetch-issue-excl-hint-
/Qopt-prefetch-next-iteration (i64 only)	Enables or disables prefetches for a memory access in the next iteration of a loop.	/Qopt-prefetch-next-iteration
/Qopt-subscript-in-range (i32, i64em)	Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.	/Qopt-subscript-in-range-
/Qpar-affinity:[modifier,...]type[,permute][,offset]	Specifies thread affinity.	OFF
/Qpar-num-threads:n	Specifies the number of threads to use in a parallel region.	OFF
/Qprof-data-order	Enables or disables data ordering if profiling information is enabled.	/Qprof-data-order
/Qprof-func-order	Enables or disables function ordering if profiling information is enabled.	/Qprof-func-order

Windows* OS Options	Description	Default
/Qprof-hot-ness-thresh-old	Lets you set the hotness threshold for function grouping and function ordering.	OFF
/Qprof-src-dir	Determines whether directory information of the source file under compilation is considered when looking up profile data records.	/Qprof-src-dir
/Qprof-src-root	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.	OFF
/Qprof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.	OFF
/Qtcollect-filter	Lets you enable or disable the instrumentation of specified functions.	OFF
/Quse-msasm-symbols (i32, i64em)	Tells the compiler to use a dollar sign ("$") when producing symbol names.	OFF
/Qvc9 (i32, i64em)	Specifies compatibility with Microsoft* Visual Studio 2008.	varies
/Qvec (i32, i64em)	Enables or disables vectorization and transformations enabled for vectorization.	/Qvec
/Qvec-thresh-old (i32, i64em)	Sets a threshold for the vectorization of loops.	/Qvec-thresh-old100
/QxHost (i32, i64em)	Can generate instructions for the highest instruction set available on the compilation host processor.	OFF
/QxAVX (i32, i64em)	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).	OFF

Windows* OS Options	Description	Default
/QxSSE2 (i32, i64em)	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2.	ON
/QxSSE3 (i32, i64em)	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors, and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture.	OFF
/QxSSE3_ATOM (i32, i64em)	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology.	OFF
/QxSSSE3 (i32, i64em)	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family.	OFF
/QxSSE4.1 (i32, i64em)	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.	OFF
/QxSSE4.2 (i32, i64em)	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.	OFF

Linux* OS and Mac OS* X Options	Description	Default
-assume [no]ieee_fpe_flags	Determines whether the floating-point exception and status flags are saved on routine entry and restored on routine exit.	OFF

Linux* OS and Mac OS* X Options	Description	Default
`-assume [no]old_logical_ldio`	Determines whether NAMELIST and list-directed input accept logical values for numeric IO-list items.	ON
`-assume [no]old_maxminloc`	Determines the results of the intrinsics MAXLOC and MINLOC when given an empty array as an argument.	ON
`-axSSE2` (i32, i64em; Linux only)	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2.	OFF
`-axSSE3` (i32, i64em)	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors, and it can optimize for processors based on Intel® Core microarchitecture and Intel NetBurst® microarchitecture. On Mac OS* X systems, this option is only available on IA-32 architecture.	OFF
`-axSSSE3` (i32, i64em)	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. On Mac OS* X systems, this option is only available on Intel® 64 architecture.	OFF
`-axSSE4.1` (i32, i64em)	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.	OFF
`-axSSE4.2` (i32, i64em)	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4	OFF

Linux* OS and Mac OS* X Options	Description	Default
	Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.	
`-diag-enable sc-parallel` (i32, i64em)	Enables analysis of parallelization in source code (parallel lint diagnostics).	OFF
`-diag-error-limit n`	Specifies the maximum number of errors allowed before compilation stops.	n=30
`-diag-once id[,id,...]`	Tells the compiler to issue one or more diagnostic messages only once.	OFF
`-falign-stack` (i32 only)	Tells the compiler the stack alignment to use on entry to routines.	`-falign-stack=default`
`-fast-tran-scendentals`	Enables the compiler to replace calls to transcendental functions with faster but less precise implementation.	OFF
`-finline`	Tells the compiler to inline functions declared with cDEC$ ATTRIBUTES FORCEINLINE.	`-fno-inline`
`-fma` (i64 only; Linux* OS only)	Enables the combining of floating-point multiplies and add/subtract operations.	ON
`-fp-relaxed` (i64 only; Linux* OS only)	Enables use of faster but slightly less accurate code sequences for math functions.	OFF
`-fpe-all=n`	Allows some control over floating-point exception handling for each routine in a program at run-time.	`-fpe-all=3`
`-fpie` (Linux* OS only)	Tells the compiler to generate position-independent code to link into executables.	OFF

Linux* OS and Mac OS* X Options	Description	Default
-fstack-protector (i32, i64em)	Determines whether the compiler generates code that detects some buffer overruns. Same as option -fstack-security-check.	-fno-stack-protector
-fstack-security-check (i32, i64em)	Determines whether the compiler generates code that detects some buffer overruns.	-fno-stack-security-check
-m32, -m64 (i32, i64em)	Tells the compiler to generate code for a specific architecture.	OFF
-mia32 (i32 only)	Generates code that will run on any Pentium or later processor.	OFF
-minstruction=[no]movbe (i32, i64em)	Determines whether MOVBE instructions are generated for Intel processors.	OFF
-mkl	Tells the compiler to link to certain parts of the Intel® Math Kernel Library.	OFF
-mssse3 (i32, i64em)	Generates code for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3).	Linux systems: OFF Mac OS X systems using Intel® 64 architecture: ON
-msse4.1 (i32, i64em)	Generates code for Intel® Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators.	OFF
-openmp-link library	Controls whether the compiler links to static or dynamic OpenMP run-time libraries.	-openmp-link dynamic
-openmp-threadprivate=type	Lets you specify an OpenMP* threadprivate implementation.	-openmp-threadprivate=legacy

Linux* OS and Mac OS* X Options	Description	Default
(Linux* OS only)		
-opt-block-factor=n	Lets you specify a loop blocking factor.	OFF
-opt-jump-tables=keyword	Enables or disables generation of jump tables for switch statements.	-opt-jump-tables=default
-opt-loadpair (i64 only; Linux* OS only)	Enables loadpair optimization.	-no-opt-load-pair
-opt-mod-versioning (i64 only; Linux* OS only)	Enables versioning of modulo operations for certain types of operands.	-no-opt-mod-versioning
-opt-prefetch-initial-values (i64 only; Linux* OS only)	Enables or disables prefetches that are issued before a loop is entered.	-opt-prefetch-initial-values
-opt-prefetch-issue-excl-hint (i64 only)	Determines whether the compiler issues prefetches for stores with exclusive hint.	-no-opt-prefetch-issue-excl-hint
-opt-prefetch-next-iteration (i64 only; Linux* OS only)	Enables or disables prefetches for a memory access in the next iteration of a loop.	-opt-prefetch-next-iteration

Linux* OS and Mac OS* X Options	Description	Default
-opt-sub-script-in-range (i32, i64em)	Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.	-no-opt-sub-script-in-range
-par-affinity=[modifier,...]type[,permute][,offset] (Linux* OS only)	Specifies thread affinity.	OFF
-par-num-threads=n	Specifies the number of threads to use in a parallel region.	OFF
-pie (Linux* OS only)	Produces a position-independent executable on processors that support it.	OFF
-prof-data-order (Linux* OS only)	Enables or disables data ordering if profiling information is enabled.	-no-prof-data-order
-prof-func-groups (i32, i64em; Linux* OS only)	Enables or disables function grouping if profiling information is enabled.	-no-prof-func-groups
-prof-func-order (Linux* OS only)	Enables or disables function ordering if profiling information is enabled.	-no-prof-func-order
-prof-hotness-threshold (Linux* OS only)	Lets you set the hotness threshold for function grouping and function ordering.	OFF

Linux* OS and Mac OS* X Options	Description	Default
-prof-src-root	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.	OFF
-prof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.	OFF
-staticlib (i32, i64em; Mac OS* X only)	Invokes the libtool command to generate static libraries.	OFF
-tcollect-filter (Linux* OS only)	Lets you enable or disable the instrumentation of specified functions.	OFF
-vec (i32, i64em)	Enables or disables vectorization and transformations enabled for vectorization.	-vec
-vec-threshold (i32, i64em)	Sets a threshold for the vectorization of loops.	-vec-threshold100
-xHost (i32, i64em)	Can generate instructions for the highest instruction set available on the compilation host processor.	OFF
-xAVX (i32, i64em)	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).	OFF
-xSSE2 (i32, i64em; Linux only)	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2.	ON

Linux* OS and Mac OS* X Options	Description	Default
-xSSE3 (i32, i64em)	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. On Mac OS* X systems, this option is only available on IA-32 architecture.	Linux systems:OFF Mac OS X systems using IA-32 architecture: ON
-xSSE3_ATOM (i32, i64em)	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology.	OFF
-xSSSE3 (i32, i64em)	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. On Mac OS* X systems, this option is only available on Intel® 64 architecture.	Linux systems:OFF Mac OS X systems using Intel® 64 architecture: ON
-xSSE4.1 (i32, i64em)	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture.	OFF
-xSSE4.2 (i32, i64em)	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.	OFF

Deprecated and Removed Compiler Options

This topic lists deprecated and removed compiler options and suggests replacement options, if any are available.

Deprecated Options

Occasionally, compiler options are marked as "deprecated." Deprecated options are still supported in the current release, but are planned to be unsupported in future releases.

The following options are deprecated in this release of the compiler:

Linux* OS and Mac OS* X Options	Suggested Replacement
`-axK`	None
`-axN`	Linux* OS: `-axSSE2` Mac OS* X: None
`-axP`	Linux* OS: `-axSSE3` Mac OS* X on IA-32 architecture: `-axSSE3` Mac OS* X on Intel® 64 architecture: None
`-axS`	`-axSSE4.1`
`-axT`	Linux* OS: `-axSSSE3` Mac OS* X on IA-32 architecture: None Mac OS* X on Intel® 64 architecture: `-axSSSE3`
`-axW`	`-msse2`
`-diag-<type> sv[<n>]`	`-diag-<type> sc[<n>]`
`-diag-enable sv-include`	`-diag-enable sc-include`
`-func-groups`	`-prof-func-groups`
`-i-dynamic`	`-shared-intel`
`-i-static`	`-static-intel`
`-inline-debug-info`	`-debug`
`-IPF-flt-eval-method0`	`-fp-model source`

Linux* OS and Mac OS* X Options	Suggested Replacement
-IPF-fltacc -no-IPF-fltacc	-fp-model precise -fp-model fast
-IPF-fma	-fma
-IPF-fp-relaxed	-fp-relaxed
-march=pentiumii	None
-march=pentiumiii	-march=pentium3
-mcpu	-mtune
-mp	-fp-model
-Ob	-inline-level
-openmp-lib legacy	None
-openmpP	-openmp
-openmpS	-openmp-stubs
-prefetch	-opt-prefetch
-prof-genx	-prof-gen=srcpos
-use-asm	None
-use-pch	-pch-use
-xK	-mia32
-xN	Linux* OS: -xSSE2 Mac OS* X: None
-xO	-msse3

Linux* OS and Mac OS* X Options	Suggested Replacement
-xP	Linux* OS: -xSSE3 Mac OS* X on IA-32 architecture: -xSSE3 Mac OS* X on Intel® 64 architecture: None
-xS	-xSSE4.1
-xT	Linux* OS: -xSSSE3 Mac OS* X on IA-32 architecture: None Mac OS* X on Intel® 64 architecture: -xSSSE3
-xW	-msse2

Windows* OS Options	Suggested Replacement
/4Nb	/check:none
/4Yb	/check:all
/debug:partial	None
/Fm	/map
/G5	None
/G6 (or /GB)	None
/G7	None
/Ge	/Gs0
/ML and/MLd	None
/Op	/fp
/QaxK	None
/QaxN	/QaxSSE2

Windows* OS Options	Suggested Replacement
/QaxP	/QaxSSE3
/QaxS	/QaxSSE4.1
/QaxT	/QaxSSSE3
/QaxW	/arch:SSE2
/Qdiag-<type> sv[<n>]	/Qdiag-<type> sc[<n>]
/Qdiag-enable:sv-include	/Qdiag-enable:sc-include
/Qinline-debug-info	None
/QIPF-flt-eval-method0	/fp:source
/QIPF-fltacc /QIPF-fltacc-	/fp:precise /fp:fast
/QIPF-fma	/Qfma
/QIPF-fp-relaxed	/Qfp-relaxed
/Qopenmp-lib:legacy	None
/Qprefetch	/Qopt-prefetch
/Qprof-genx	/Qprof-gen=srcpos
/Quse-asm	None
/Quse-vcdebug	None
/QxK	None
/QxN	/QxSSE2
/QxO	/arch:SSE3

Windows* OS Options	Suggested Replacement
/QxP	/QxSSE3
/QxS	/QxSSE4.1
/QxT	/QxSSSE3
/QxW	/arch:SSE2
/Zd	/debug:minimal

Deprecated options are not limited to this list.

Removed Options

Some compiler options are no longer supported and have been removed. If you use one of these options, the compiler issues a warning, ignores the option, and then proceeds with compilation.

This version of the compiler no longer supports the following compiler options:

Linux* OS and Mac OS* X Options	Suggested Replacement
-axB	-axSSE2
-axi	None
-axM	None
-cxxlib-gcc[=dir]	-cxxlib[=dir]
-cxxlib-icc	None
-F	-preprocess-only or -P
-fp	-fno-omit-frame-pointer
-fpstkchk	-fp-stack-check
-IPF-fp-speculation	-fp-speculation

Linux* OS and Mac OS* X Options	Suggested Replacement
-ipo-obj (and -ipo_obj)	None
-Kpic, -KPIC	-fpic
-mtune=itanium	None
-nobss-init	-no-bss-init
-opt-report-level	-opt-report
-prof-format-32	None
-prof-gen-sampling	None
-qp	-p
-shared-libcxa	-shared-libgcc
-ssp	None
-static-libcxa	-static-libgcc
-syntax	-syntax-only or -fsyntax-only
-tpp1	None
-tpp2	-mtune=itanium2
-tpp5	None
-tpp6	None
-tpp7	-mtune=pentium4
-xB	-xSSE2
-xi	None
-xM	None

Windows* OS Options	Suggested Replacement
/4ccD (and /4ccd)	None
/G1	None
/QaxB	/QaxSSE2
/Qaxi	None
/QaxM	None
/Qfpstkchk	/Qfp-stack-check
/QIPF-fp-speculation	/Qfp-speculation
/Qipo-obj (and /Qipo_obj)	None
/Qopt-report-level	/Qopt-report
/Qprof-format-32	None
/Qprof-gen-sampling	None
/Qssp	None
/Qvc6	None
/Qvc7	None
/QxB	/QxSSE2
/Qxi	None
/QxM	None

Removed options are not limited to these lists.

Alphabetical Compiler Options 20

Compiler Option Descriptions and General Rules

This section describes all the current Linux* OS, Mac OS* X, and Windows* OS compiler options in alphabetical order.

Option Descriptions

Each option description contains the following information:

- A short description of the option.
- IDE Equivalent
  This shows information related to the integrated development environment (IDE) Property Pages on Windows*, Linux*, and Mac OS* X systems. It shows on which Property Page the option appears, and under what category it's listed. The Windows IDE is Microsoft* Visual Studio* .NET; the Linux IDE is Eclipse*; the Mac OS X IDE is Xcode*. If the option has no IDE equivalent, it will specify "None". Note that in this release, there is no IDE support for Fortran on Linux.
- Architectures
  This shows the architectures where the option is valid. Possible architectures are:
  - IA-32 architecture
  - Intel® 64 architecture
  - IA-64 architecture
- Syntax
  This shows the syntax on Linux and Mac OS X systems and the syntax on Windows systems. If the option has no syntax on one of these systems, that is, the option is not valid on a particular system, it will specify "None".
- Arguments
  This shows any arguments (parameters) that are related to the option. If the option has no arguments, it will specify "None".
- Default
  This shows the default setting for the option.
- Description

This shows the full description of the option. It may also include further information on any applicable arguments.

- Alternate Options
  These are options that are synonyms of the described option. If there are no alternate options, it will specify "None".
  Many options have an older spelling where underscores ("_") instead of hyphens ("-") connect the main option names. The older spelling is a valid alternate option name.

Some option descriptions may also have the following:

- Example
  This shows a short example that includes the option
- See Also
  This shows where you can get further information on the option or related options.

General Rules for Compiler Options

You cannot combine options with a single dash (Linux OS and Mac OS X) or slash (Windows OS). For example:

- On Linux and Mac OS X systems: This is incorrect: `-wc`; this is correct: `-w -c`
- On Windows systems: This is incorrect: `/wc`; this is correct: `/w /c`

All Linux OS and Mac OS X compiler options are case sensitive. Many Windows OS options are case sensitive. Some options have different meanings depending on their case; for example, option "c" prevents linking, but option "C" checks for certain conditions at run time.

Options specified on the command line apply to all files named on the command line.

Options can take arguments in the form of file names, strings, letters, or numbers. If a string includes spaces, the string must be enclosed in quotation marks. For example:

- On Linux and Mac OS X systems, `-dynamic-linker` *`mylink`* (file name) or `-U`*`macro3`* (string)
- On Windows systems, `/Famyfile.s` (file name) or `/V"`*`version 5.0`*`"` (string)

Compiler options can appear in any order.

On Windows systems, all compiler options must precede `/link` options, if any, on the command line.

Unless you specify certain options, the command line will both compile and link the files you specify.

You can abbreviate some option names, entering as many characters as are needed to uniquely identify the option.

Certain options accept one or more keyword arguments following the option name. For example, the `arch` option accepts several keywords.

To specify multiple keywords, you typically specify the option multiple times. However, there are exceptions; for example, the following are valid: `-axNB` (Linux OS) or `/QaxNB` (Windows OS).

NOTE. On Windows systems, you can sometimes use a comma to separate keywords. For example, the following is valid:

```
ifort /warn:usage,declarations test.f90
```

On these systems, you can use an equals sign (=) instead of the colon:

```
ifort /warn=usage,declarations test.f90
```

Compiler options remain in effect for the whole compilation unless overridden by a compiler directive.

To disable an option, specify the negative form of the option.

On Windows systems, you can also disable one or more options by specifying option `/Od` last on the command line.

NOTE. On Windows systems, the `/Od` option is part of a mutually-exclusive group of options that includes `/Od`, `/O1`, `/O2`, `/O3`, and `/Ox`. The last of any of these options specified on the command line will override the previous options from this group.

If there are enabling and disabling versions of an option on the command line, the last one on the command line takes precedence.

Lists and Functional Groupings of Compiler Options

To see a list of all the compiler options, specify option `help` on the command line.

To see functional groupings of compiler options, specify a functional category for option `help`. For example, to see a list of options that affect diagnostic messages displayed by the compiler, enter one of the following commands:

```
-help diagnostics     ! Linux and Mac OS X systems

/help diagnostics     ! Windows systems
```

For details on the categories you can specify, see help.

1

See onetrip.

4I2, 4I4, 4I8

See integer-size.

4L72, 4L80, 4L132

See extend-source.

4Na, 4Ya

See automatic.

4Naltparam, 4Yaltparam

See altparam.

4Nb,4Yb

See check.

4Nd,4Yd

See warn.

4Nf

See fixed.

4Nportlib, 4Yportlib

Determines whether the compiler links to the library of portability routines.

IDE Equivalent

Windows: **Libraries > Use Portlib Library**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/4Nportlib`

`/4Yportlib`

Arguments

None

Default

`/4Yportlib`	The library of portability routines is linked during compilation.

Description

Option `/4Yportlib` causes the compiler to link to the library of portability routines. This also includes Intel's functions for Microsoft* compatibility.

Option `/4Nportlib` prevents the compiler from linking to the library of portability routines.

Alternate Options

None

See Also

-
-

Building Applications: Portability Routines

4Ns,4Ys

See stand.

4R8,4R16

See real-size.

4Yf

See free.

4Nportlib, 4Yportlib

Determines whether the compiler links to the library of portability routines.

IDE Equivalent

Windows: **Libraries > Use Portlib Library**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/4Nportlib`

`/4Yportlib`

Arguments

None

Default

`/4Yportlib`	The library of portability routines is linked during compilation.

Description

Option `/4Yportlib` causes the compiler to link to the library of portability routines. This also includes Intel's functions for Microsoft* compatibility.

Option `/4Nportlib` prevents the compiler from linking to the library of portability routines.

Alternate Options

None

See Also

•
•
Building Applications: Portability Routines

66

See f66.

72,80,132

See extend-source.

align

Tells the compiler how to align certain data items.

IDE Equivalent

Windows: **Data > Structure Member Alignment** (`/align:rec`*n*`byte`)

Data > Common Element Alignment (`/align:[no]commons`, `/align:[no]dcommons`)

Data > SEQUENCE Types Obey Alignment Rules (`/align:[no]sequence`)

Linux: None

Mac OS X: **Data > Structure Member Alignment** (`-align rec<1,2,4,8,16>byte`)

Data > Common Element Alignment (`-align [no]commons`, `/align:[no]dcommons`)

Data > SEQUENCE Types Obey Alignment Rules (`-align [no]sequence`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-align [keyword]`

`-noalign`

Windows:

`/align[:keyword]`

`/noalign`

Arguments

keyword Specifies the data items to align. Possible values are:

`none`	Prevents padding bytes anywhere in common blocks and structures.
`[no]commons`	Affects alignment of common block entities.
`[no]dcommons`	Affects alignment of common block entities.
`[no]records`	Affects alignment of derived-type components and fields of record structures.
`rec`*n*`byte`	Specifies a size boundary for derived-type components and fields of record structures.
`[no]sequence`	Affects alignment of sequenced derived-type components.
`all`	Adds padding bytes whenever possible to data items in common blocks and structures.

Default

`nocommons`	Adds no padding bytes for alignment of common blocks.
`nodcommmons`	Adds no padding bytes for alignment of common blocks.
`records`	Aligns derived-type components and record structure fields on default natural boundaries.

`nosequence`	Causes derived-type components declared with the SEQUENCE statement to be packed, regardless of current alignment rules set by the user.

By default, no padding is added to common blocks but padding is added to structures.

Description

This option specifies the alignment to use for certain data items. The compiler adds padding bytes to perform the alignment.

Option	Description
`align none`	Tells the compiler not to add padding bytes anywhere in common blocks or structures. This is the same as specifying `noalign`.
`align commons`	Aligns all common block entities on natural boundaries up to 4 bytes, by adding padding bytes as needed. The `align nocommons` option adds no padding to common blocks. In this case, unaligned data can occur unless the order of data items specified in the COMMON statement places the largest numeric data item first, followed by the next largest numeric data (and so on), followed by any character data.
`align dcommons`	Aligns all common block entities on natural boundaries up to 8 bytes, by adding padding bytes as needed. This option is useful for applications that use common blocks, unless your application has no unaligned data or, if the application might have unaligned data, all data items are four bytes or smaller. For applications that use common blocks where all data items are four bytes or smaller, you can specify `/align:commons` instead of `/align:dcommons`. The `align nodcommons` option adds no padding to common blocks. On Windows systems, if you specify the `/stand:f90` or `/stand:f95` option, `/align:dcommons` is ignored. On Linux and Mac OS X systems, if you specify any `-std` option or the `-stand f90` or `-stand f95` option, `-align dcommons` is ignored.
`align norecords`	Aligns components of derived types and fields within record structures on arbitrary byte boundaries with no padding.

Option	Description
	The align records option requests that multiple data items in record structures and derived-type structures without the SEQUENCE statement be naturally aligned, by adding padding as needed.
`align rec`*`n`*`byte`	Aligns components of derived types and fields within record structures on the smaller of the size boundary specified (*`n`*) or the boundary that will naturally align them. *`n`* can be 1, 2, 4, 8, or 16. When you specify this option, each structure member after the first is stored on either the size of the member type or *`n`*-byte boundaries, whichever is smaller. For example, to specify 2 bytes as the packing boundary (or alignment constraint) for all structures and unions in the file prog1.f, use the following command: `ifort {-align rec2byte | /align:rec2byte} prog1.f` This option does not affect whether common blocks are naturally aligned or packed.
`align sequence`	Aligns components of a derived type declared with the SEQUENCE statement (sequenced components) according to the alignment rules that are currently in use. The default alignment rules are to align unsequenced components on natural boundaries. The `align nosequence` option requests that sequenced components be packed regardless of any other alignment rules. Note that `align none` implies `align nosequence`. If you specify an option for standards checking, `/align:sequence` is ignored.
`align all`	Tells the compiler to add padding bytes whenever possible to obtain the natural alignment of data items in common blocks, derived types, and record structures. Specifies `align nocommons, align dcommons, align records, align nosequence`. This is the same as specifying `align` with no *`keyword`*.

Alternate Options

`align none`	Linux and Mac OS X: `-noalign` Windows: `/noalign`

`align records`	Linux and Mac OS X: `-align rec16byte`, `-Zp16` Windows: `/align:rec16byte`, `/Zp16`
`align norecords`	Linux and Mac OS X: `-Zp1`, `-align rec1byte` Windows: `/Zp1`, `/align:rec1byte`
`align rec`*n*`byte`	Linux and Mac OS X: `-Zp{1|2|4|8|16}` Windows: `/Zp{1|2|4|8|16}`
`align all`	Linux and Mac OS X: `-align commons -align dcommons -align records -align nosequence` Windows: `/align:nocommons,dcommons,records,nosequence`

See Also

- Optimizing Applications: Setting Data Type and Alignment

allow

Determines whether the compiler allows certain behaviors.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-allow` *keyword*

Windows:

`/allow:`*keyword*

Arguments

keyword	Specifies the behaviors to allow or disallow. Possible values are:

`[no]fpp_comments`	Determines how the fpp preprocessor treats Fortran end-of-line comments in preprocessor directive lines.

Default

`fpp_comments`	The compiler recognizes Fortran-style end-of-line comments in preprocessor lines.

Description

This option determines whether the compiler allows certain behaviors.

Option	Description
`allow nofpp_comments`	Tells the compiler to disallow Fortran-style end-of-line comments on preprocessor lines. Comment indicators have no special meaning.

Alternate Options

None

Example

Consider the following:

```
#define MAX_ELEMENTS 100  ! Maximum number of elements
```

By default, the compiler recognizes Fortran-style end-of-line comments on preprocessor lines. Therefore, the line above defines MAX_ELEMENTS to be "100" and the rest of the line is ignored. If `allow nofpp_comments` is specified, Fortran comment conventions are not used and the comment indicator "!" has no special meaning. So, in the above example, "! Maximum number of elements" is interpreted as part of the value for the MAX_ELEMENTS definition.

Option `allow nofpp_comments` can be useful when you want to have a Fortran directive as a define value; for example:

```
#define dline(routname) !dec$ attributes alias:"__routname":: routname
```

altparam

Allows alternate syntax (without parentheses) for PARAMETER statements.

IDE Equivalent

Windows: **Language > Enable Alternate PARAMETER Syntax**

Linux: None

Mac OS X: **Language > Enable Alternate PARAMETER Syntax**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-altparam
-noaltparam
```

Windows:

```
/altparam
/noaltparam
```

Arguments

None

Default

`altparam`	The alternate syntax for PARAMETER statements is allowed.

Description

This option specifies that the alternate syntax for PARAMETER statements is allowed. The alternate syntax is:

```
PARAMETER c = expr [, c = expr] ...
```

This statement assigns a name to a constant (as does the standard PARAMETER statement), but there are no parentheses surrounding the assignment list.

In this alternative statement, the form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

Alternate Options

`altparam`	Linux and Mac OS X: `-dps` Windows: `/Qdps`, `/4Yaltparam`
`noaltparam`	Linux and Mac OS X: `-nodps` Windows: `/Qdps-`, `/4Naltparam`

ansi-alias, Qansi-alias

Tells the compiler to assume that the program adheres to Fortran Standard type aliasability rules.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ansi-alias`

`-no-ansi-alias`

Windows:

`Qansi-alias`

`Qansi-alias-`

Arguments

None

Default

`-ansi-alias` or `/Qansi-alias`	Programs adhere to Fortran Standard type aliasability rules.

Description

This option tells the compiler to assume that the program adheres to type aliasability rules defined in the Fortran Standard.

For example, an object of type real cannot be accessed as an integer. For information on the rules for data types and data type constants, see "Data Types, Constants, and Variables" in the Language Reference.

This option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.
- References to objects of two different scalar types cannot alias. For example, an object of type integer cannot alias with an object of type real or an object of type real cannot alias with an object of type double precision.

If your program adheres to the Fortran Standard type aliasability rules, this option enables the compiler to optimize more aggressively. If it doesn't adhere to these rules, then you should disable the option with `-no-ansi-alias` (Linux and Mac OS X) or `/Qansi-alias-` (Windows) so the compiler does not generate incorrect code.

Alternate Options

None

arch

Tells the compiler to generate optimized code specialized for the processor that executes your program.

IDE Equivalent

Windows: **Code Generation > Enable Enhanced Instruction Set**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-arch`*`processor`*

Windows:

`/arch:`*`processor`*

Arguments

`processor` Is the processor type. Possible values are:

`IA32`	Generates code that will run on any Pentium or later processor. Disables any default extended instruction settings, and any previously set extended instruction settings. This value is only available on Linux and Windows systems using IA-32 architecture.
`SSE`	This is the same as specifying `IA32`.
`SSE2`	Generates code for Intel® Streaming SIMD Extensions 2 (Intel® SSE2). This value is only available on Linux and Windows systems.
`SSE3`	Generates code for Intel® Streaming SIMD Extensions 3 (Intel® SSE3).
`SSSE3`	Generates code for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3).
`SSE4.1`	Generates code for Intel® Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators.

Default

Windows and Linux systems: For more information on the default values, see Arguments above.
`SSE2`
Mac OS X systems using
IA-32 architecture: `SSE3`

Mac OS X systems using
Intel® 64 architecture:
SSSE3

Description

This option tells the compiler to generate optimized code specialized for the processor that executes your program.

Code generated with the values IA32, SSE, SSE2, or SSE3 should execute on any compatible non-Intel processor with support for the corresponding instruction set.

Options /arch and /Qx are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

For compatibility with Compaq* Visual Fortran, the compiler allows the following keyword values. However, you should use the suggested replacements.

Compatibility Value	Suggested Replacement
pn1	-mia32 or /arch:IA32
pn2	-mia32 or /arch:IA32
pn3	-mia32 or /arch:IA32
pn4	-msse2 or /arch:SSE2

Alternate Options

Linux and Mac OS X: -m

Windows: /architecture

See Also

-
- x, Qx
- ax, Qax
- m

architecture

See arch.

asmattr

Specifies the contents of an assembly listing file.

IDE Equivalent

Windows: **Output > Assembler Output**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/asmattr:`*`keyword`*

`/noasmattr`

Arguments

keyword	Specifies the contents of the assembly listing file. Possible values are:	
	`none`	Produces no assembly listing.
	`machine`	Produces an assembly listing with machine code.
	`source`	Produces an assembly listing with source code.
	`all`	Produces an assembly listing with machine code and source code.

Default

/noasmattr	No assembly listing is produced.

Description

This option specifies what information, in addition to the assembly code, should be generated in the assembly listing file.

To use this option, you must also specify option /asmfile, which causes an assembly listing to be generated.

Option	Description
/asmattr:none	Produces no assembly listing. This is the same as specifying /noasmattr.
/asmattr:machine	Produces an assembly listing with machine code. The assembly listing file shows the hex machine instructions at the beginning of each line of assembly code. The file cannot be assembled; the filename is the name of the source file with an extension of .cod.
/asmattr:source	Produces an assembly listing with source code. The assembly listing file shows the source code as interspersed comments. Note that if you use alternate option -fsource-asm, you must also specify the -S option.
/asmattr:all	Produces an assembly listing with machine code and source code. The assembly listing file shows the source code as interspersed comments and shows the hex machine instructions at the beginning of each line of assembly code. This file cannot be assembled.

Alternate Options

/asmattr:none	Linux and Mac OS X: None Windows: /noasmattr
/asmattr:machine	Linux and Mac OS X: -fcode-asm Windows: /FAc
/asmattr:source	Linux and Mac OS X: -fsource-asm Windows: /FAs

`/asmattr:all`	Linux and Mac OS X: None Windows: `/FAcs`

See Also

-
- `asmfile`

asmfile

Specifies that an assembly listing file should be generated.

IDE Equivalent

Windows: **Output > ASM Listing Name**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/asmfile[:`*`file`* `|` *`dir`*`]`

`/noasmfile`

Arguments

`file`	Is the name of the assembly listing file.
`dir`	Is the directory where the file should be placed. It can include *`file`*.

Default

`/noasmfile`	No assembly listing file is produced.

Description

This option specifies that an assembly listing file should be generated (optionally named *file*).

If *file* is not specified, the filename will be the name of the source file with an extension of .asm; the file is placed in the current directory.

Alternate Options

Linux and Mac OS X: -S

Windows: /Fa

See Also

-
- S

assume

Tells the compiler to make certain assumptions.

IDE Equivalent

Windows: **Compatibility > Treat Backslash as Normal Character in Strings** (/assume:[no]bscc)

Data > Assume Dummy Arguments Share Memory Locations (/assume:[no]dummy_aliases)

Data > Constant Actual Arguments Can Be Changed (/assume:[no]protect_constants)

Data > Use Bytes as RECL=Unit for Unformatted Files (/assume:[no]byterecl)

Floating Point > Enable IEEE Minus Zero Support (/assume:[no]minus0)

Optimization > I/O Buffering (/assume:[no]buffered_io)

Preprocessor > Default Include and Use Path (/assume:[no]source_include)

Preprocessor > OpenMP Conditional Compilation (/assume:[no]cc_omp)

External Procedures > Append Underscore to External Names (/assume:[no]underscore)

Linux: None

Mac OS X: **Optimization > I/O Buffering** (-assume [no]buffered_io)

Preprocessor > OpenMP Conditional Compilation (`-assume [no]cc_omp`)

Preprocessor > Default Include and Use Path (`-assume [no]source_include`)

Compatibility > Treat Backslash as Normal Character in Strings (`-assume [no]bscc`)

Data > Assume Dummy Arguments Share Memory Locations (`-assume [no]dummy_aliases`)

Data > Constant Actual Arguments Can Be Changed (`-assume [no]protect_constants`)

Data > Use Bytes as RECL=Unit for Unformatted Files (`-assume [no]byterecl`)

Floating Point > Enable IEEE Minus Zero Support (`-assume [no]minus0`)

External Procedures > Append Underscore to External Names (`-assume [no]underscore`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-assume` *keyword*

Windows:

`/assume:`*keyword*

Arguments

keyword Specifies the assumptions to be made. Possible values are:

`none`	Disables all assume options.
`[no]bscc`	Determines whether the backslash character is treated as a C-style control character syntax in character literals.
`[no]buffered_io`	Determines whether data is immediately written to disk or accumulated in a buffer.
`[no]byterecl`	Determines whether units for the OPEN statement RECL specifier (record length) value in unformatted files are in bytes or longwords (four-byte units).

[no]cc_omp	Determines whether conditional compilation as defined by the OpenMP Fortran API is enabled or disabled.
[no]dummy_aliases	Determines whether the compiler assumes that dummy arguments to procedures share memory locations with other dummy arguments or with COMMON variables that are assigned.
[no]ieee_fpe_flags	Determines whether the floating-point exception and status flags are saved on routine entry and restored on routine exit.
[no]minus0	Determines whether the compiler uses Fortran 95 or Fortran 90/77 standard semantics in the SIGN intrinsic when treating -0.0 and +0.0 as 0.0, and how it writes the value on formatted output.
[no]old_boz	Determines whether the binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX are treated as signed integer constants.
[no]old_logical_ldio	Determines whether NAMELIST and list-directed input accept logical values for numeric IO-list items.
[no]old_maxminloc	Determines the results of intrinsics MAXLOC and MINLOC when given an empty array as an argument.
[no]old_unit_star	Determines whether READs or WRITEs to UNIT=* go to stdin or stdout, respectively.
[no]old_xor	Determines whether .XOR. is defined by the compiler as an intrinsic operator.
[no]protect_constants	Determines whether a constant actual argument or a copy of it is passed to a called routine.

[no]protect_parens	Determines whether the optimizer honors parentheses in REAL and COMPLEX expression evaluations by not reassociating operations.
[no]realloc_lhs	Determines whether allocatable objects on the left-hand side of an assignment are treated according to Fortran 95/90 rules or Fortran 2003 rules.
[no]source_include	Determines whether the compiler searches for USE modules and INCLUDE files in the default directory or in the directory where the source file is located.
[no]std_mod_proc_name	Determines whether the names of module procedures are allowed to conflict with user external symbol names.
[no]underscore	Determines whether the compiler appends an underscore character to external user-defined names.
[no]2underscores (Linux and Mac OS X)	Determines whether the compiler appends two underscore characters to external user-defined names.
[no]writeable_strings	Determines whether character constants go into non-read-only memory.

Default

nobscc	The backslash character is treated as a normal character in character literals.
nobuffered_io	Data in the internal buffer is immediately written (flushed) to disk (OPEN specifier BUFFERED='NO'). If you set the FORT_BUFFERED environment variable to true, the default is `assume buffered_io`.
nobyterecl	Units for OPEN statement RECL values with unformatted files are in four-byte (longword) units.

nocc_omp	Conditional compilation as defined by the OpenMP Fortran API is disabled unless option -openmp (Linux) or /Qopenmp (Windows) is specified. If compiler option -openmp (Linux and Mac OS X) or /Qopenmp (Windows) is specified, the default is assume cc_omp.
nodummy_aliases	Dummy arguments to procedures do not share memory locations with other dummy arguments or with variables shared through use association, host association, or common block use.
noieee_fpe_flags	The flags are not saved on routine entry and they are not restored on routine exit.
nominus0	The compiler uses Fortran 90/77 standard semantics in the SIGN intrinsic to treat -0.0 and +0.0 as 0.0, and writes a value of 0.0 with no sign on formatted output.
noold_boz	The binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX are treated as bit strings that represent a value of the data type of the intrinsic, that is, the bits are not converted.
noold_logical_ldio	Tells the compiler that NAMELIST and list-directed input cannot accept logical values (T, F, etc.) for numeric (integer, real, and complex) IO-list items. If this option is specified and a logical value is given for a numeric item in NAMELIST and list-directed input, a runtime error will be produced.
old_maxminloc	MAXLOC and MINLOC return 1 when given an empty array as an argument.
noold_unit_star	The READs or WRITEs to UNIT=* go to stdin or stdout, respectively, even if UNIT=5 or 6 has been connected to another file.
old_xor	Intrinsic operator .XOR. is defined by the compiler.
protect_constants	A constant actual argument is passed to a called routine. Any attempt to modify it results in an error.
noprotect_parens	The optimizer reorders REAL and COMPLEX expressions without regard for parentheses if it produces faster executing code.
norealloc_lhs	The compiler uses Fortran 95/90 rules when interpreting assignment statements. The left-hand side is assumed to be allocated with the correct shape to hold the right-hand side. If it is not, incorrect behavior will occur.
source_include	The compiler searches for USE modules and INCLUDE files in the directory where the source file is located.

`nostd_mod_proc_name`	The compiler allows the names of module procedures to conflict with user external symbol names.
Windows: `nounderscore` Linux and Mac OS X: `under-score`	On Windows systems, the compiler does not append an underscore character to external user-defined names. On Linux and Mac OS X systems, the compiler appends an underscore character to external user-defined names.
`no2underscores` (Linux and Mac OS X)	The compiler does not append two underscore characters to external user-defined names that contain an embedded underscore.
`nowriteable-strings`	The compiler puts character constants into read-only memory.

Description

This option specifies assumptions to be made by the compiler.

Option	Description
`assume none`	Disables all the assume options.
`assume bscc`	Tells the compiler to treat the backslash character (\) as a C-style control (escape) character syntax in character literals. The "bscc" keyword means "BackSlashControlCharacters."
`assume buffered_io`	Tells the compiler to accumulate records in a buffer. This sets the default for opening sequential output files to BUFFERED='YES', which also occurs if the FORT_BUFFERED run-time environment variable is specified. When this option is specified, the internal buffer is filled, possibly by many record output statements (WRITE), before it is written to disk by the Fortran run-time system. If a file is opened for direct access, I/O buffering is ignored. Using buffered writes usually makes disk I/O more efficient by writing larger blocks of data to the disk less often. However, if you request buffered writes, records not yet written to disk may be lost in the event of a system failure. The OPEN statement BUFFERED specifier applies to a specific logical unit. In contrast, the `assume [no]buffered_io` option and the FORT_BUFFERED environment variable apply to all Fortran units.

Option	Description
`assume byterecl`	Specifies that the units for the OPEN statement RECL specifier (record length) value are in bytes for unformatted data files, not longwords (four-byte units). For formatted files, the RECL value is always in bytes. If a file is open for unformatted data and `assume byterecl` is specified, INQUIRE returns RECL in bytes; otherwise, it returns RECL in longwords. An INQUIRE returns RECL in bytes if the unit is not open.
`assume cc_omp`	Enables conditional compilation as defined by the OpenMP Fortran API. That is, when "!$space" appears in free-form source or "c$spaces" appears in column 1 of fixed-form source, the rest of the line is accepted as a Fortran line.
`assume dummy_aliases`	Tells the compiler that dummy (formal) arguments to procedures share memory locations with other dummy arguments (aliases) or with variables shared through use association, host association, or common block use. Specify the option when you compile the called subprogram. The program semantics involved with dummy aliasing do not strictly obey the Fortran 95/90 standards and they slow performance, so you get better run-time performance if you do not use this option. However, if a program depends on dummy aliasing and you do not specify this option, the run-time behavior of the program will be unpredictable. In such programs, the results will depend on the exact optimizations that are performed. In some cases, normal results will occur, but in other cases, results will differ because the values used in computations involving the offending aliases will differ.
`assume ieee_fpe_flags`	Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance because it provides extra code to save and restore the floating-point exception and status flags (and the rounding mode) on entry to and exit from every routine compiled with the option. This option can be used to get the full Fortran Standard behavior of intrinsic modules IEEE EXCEPTIONS, IEEE ARITHMETIC, and IEEE FEATURES, which require that if a flag is signaling on routine entry, the

Option	Description
	processor will set it to quiet on entry and restore it to signaling on return. If a flag signals while the routine is executing, it will not be set to quiet on routine exit. Options `fpe` and `fpe-all` can be used to set the initial state for which floating-point exceptions will signal.
`assume minus0`	Tells the compiler to use Fortran 95 standard semantics for the treatment of the IEEE* floating value -0.0 in the SIGN intrinsic, which distinguishes the difference between -0.0 and +0.0, and to write a value of -0.0 with a negative sign on formatted output.
`assume old_boz`	Tells the compiler that the binary, octal, and hexadecimal constant arguments in intrinsic functions INT, REAL, DBLE, and CMPLX should be treated as signed integer constants.
`assume old_logical_ldio`	Logical values are allowed for numeric items.
`assume noold_maxminloc`	Tells the compiler that MAXLOC and MINLOC should return 0 when given an empty array as an argument. Compared to the default setting (`old_maxminloc`), this behavior may slow performance because of the extra code needed to check for an empty array argument.
`assume old_unit_star`	Tells the compiler that READs or WRITEs to UNIT=* go to whatever file UNIT=5 or 6 is connected.
`assume noold_xor`	Prevents the compiler from defining .XOR. as an intrinsic operator. This lets you use .XOR. as a user-defined operator. This is a Fortran 2003 feature.
`assume noprotect_constants`	Tells the compiler to pass a copy of a constant actual argument. This copy can be modified by the called routine, even though the Fortran standard prohibits such modification. The calling routine does not see any modification to the constant.

Option	Description
`assume protect_parens`	Tells the optimizer to honor parentheses in REAL and COMPLEX expression evaluations by not reassociating operations. For example, (A+B)+C would not be evaluated as A+(B+C). If assume noprotect_parens is specified, (A+B)+C would be treated the same as A+B+C and could be evaluated as A+(B+C) if it produced faster executing code. Such reassociation could produce different results depending on the sizes and precision of the arguments. For example, in (A+B)+C, if B and C had opposite signs and were very large in magnitude compared to A, A+B could result in the value as B; adding C would result in 0.0. With reassociation, B+C would be 0.0; adding A would result in a non-zero value.
`assume realloc_lhs`	Tells the compiler that when the left-hand side of an assignment is an allocatable object, it should be reallocated to the shape of the right-hand side of the assignment before the assignment occurs. This is the Fortran 2003 definition. This feature may cause extra overhead at run time.
`assume nosource_include`	Tells the compiler to search the default directory for module files specified by a USE statement or source files specified by an INCLUDE statement. This option affects fpp preprocessor behavior and the USE statement.
`assume std_mod_proc_name`	Tells the compiler to revise the names of module procedures so they do not conflict with user external symbol names. For example, procedure `proc` in module *`m`* would be named `m_MP_proc`. The Fortran 2003 Standard requires that module procedure names not conflict with other external symbols. By default, procedure `proc` in module m would be named `m_mp_proc`, which could conflict with a user-defined external name `m_mp_proc`.
`assume underscore`	Tells the compiler to append an underscore character to external user-defined names: the main program name, named common blocks, BLOCK DATA blocks, global data names in MODULEs, and names implicitly or explicitly declared EXTERNAL. The name of a blank (unnamed) common block remains _BLNK__, and Fortran intrinsic names are not affected.

Option	Description
`assume 2underscores` (Linux and Mac OS X)	Tells the compiler to append two underscore characters to external user-defined names that contain an embedded underscore: the main program name, named common blocks, BLOCK DATA blocks, global data names in MODULEs, and names implicitly or explicitly declared EXTERNAL. The name of a blank (unnamed) common block remains _BLNK__, and Fortran intrinsic names are not affected. This option does not affect external names that do not contain an embedded underscore. By default, the compiler only appends one underscore to those names. For example, if you specify `assume 2underscores` for external names *my_program* and *myprogram*, *my_program* becomes *my_program__*, but *myprogram* becomes *myprogram_*.
`assume writeable-strings`	Tells the compiler to put character constants into non-read-only memory.

Alternate Options

`assume nobscc`	Linux and Mac OS X: `-nbs` Windows: `/nbs`
`assume dummy_aliases`	Linux and Mac OS X: `-common-args` Windows: `/Qcommon-args`
`assume underscore`	Linux and Mac OS X: `-us` Windows: `/us`
`assume nounderscore`	Linux and Mac OS X: `-nus` Windows: None

See Also

-
- `fpe`
- `fpe-all`

auto, Qauto

See automatic.

auto-scalar, Qauto-scalar

Causes scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute to be allocated to the run-time stack.

IDE Equivalent

Windows: **Data > Local Variable Storage** (`/Qsave`, `/Qauto`, `/Qauto_scalar`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-auto-scalar`

Windows:

`/Qauto-scalar`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute are allocated to the run-time stack. Note that if option `recursive`, `-openmp` (Linux and Mac OS X), or `/Qopenmp` (Windows) is specified, the default is `automatic`.

Description

This option causes allocation of scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL to the run-time stack. It is as if they were declared with the AUTOMATIC attribute.

It does not affect variables that have the SAVE attribute (which include initialized locals) or that appear in an EQUIVALENCE statement or in a common block.

This option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a SAVE statement.

You cannot specify option `save`, `auto`, or `automatic` with this option.

NOTE. On Windows NT* systems, there is a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/automatic`, `/auto`, or `/Qauto` because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty. `/Qauto-scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

Alternate Options

None

See Also

-
-
- `auto`
- `save`

autodouble, Qautodouble

See real-size.

automatic

Causes all local, non-SAVEd variables to be allocated to the run-time stack.

IDE Equivalent

Windows: **Data > Local Variable Storage**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-automatic`

`-noautomatic`

Windows:

`/automatic`

`/noautomatic`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL are allocated to the run-time stack. Note that if one of the following options are specified, the default is `automatic:recursive`, `-openmp` (Linux and Mac OS X), or `/Qopenmp` (Windows).

Description

This option places local variables (scalars and arrays of all types), except those declared as SAVE, on the run-time stack. It is as if the variables were declared with the AUTOMATIC attribute.

It does not affect variables that have the SAVE attribute or ALLOCATABLE attribute, or variables that appear in an EQUIVALENCE statement or in a common block.

This option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly.

If you want to cause variables to be placed in static memory, specify option `-save` (Linux and Mac OS X) or `/Qsave` (Windows). If you want only scalar variables of certain intrinsic types to be placed on the run-time stack, specify option `auto-scalar`.

NOTE. On Windows NT* systems, there is a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/automatic`, `/auto`, or `/Qauto` because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty. `/Qauto-scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

Alternate Options

`automatic`	Linux and Mac OS X: `-auto` Windows: `/auto`, `/Qauto`, `/4Ya`
`noautomatic`	Linux and Mac OS X: `-save`, `-noauto` Windows: `/Qsave`, `/noauto`, `/4Na`

See Also

-
- `auto-scalar`
- `save`, `Qsave`

ax, Qax

Tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit.

IDE Equivalent

Windows: **Code Generation > Add Processor-Optimized Code Path**

Optimization > Generate Alternate Code Paths

Linux: None

Mac OS X: **Code Generation > Add Processor-Optimized Code Path**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-ax`*`processor`*

Windows:

`/Qax`*`processor`*

Arguments

`processor`

Indicates the processor for which code is generated. The following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`SSE4.2`

Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.

`SSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.
`SSSE3`	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. For Mac OS* X systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. For Mac OS* X systems, this value is only supported on IA-32 architecture. This replaces value P, which is deprecated.
`SSE2`	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. This value is not available on Mac OS* X systems. This replaces value N, which is deprecated.

Default

OFF	No auto-dispatch code is generated. Processor-specific code is generated and is controlled by the setting of compiler option `-m` (Linux), compiler option `/arch` (Windows), or compiler option `-x` (Mac OS* X).

Description

This option tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit. It also generates a baseline code path. The baseline code is usually slower than the specialized code.

The baseline code path is determined by the architecture specified by the `-x` (Linux and Mac OS X) or `/Qx` (Windows) option. While there are defaults for the `-x` or `/Qx` option that depend on the operating system being used, you can specify an architecture for the baseline code that is higher or lower than the default. The specified architecture becomes the effective minimum architecture for the baseline code path.

If you specify both the `-ax` and `-x` options (Linux and Mac OS X) or the `/Qax` and `/Qx` options (Windows), the baseline code will only execute on processors compatible with the processor type specified by the `-x` or `/Qx` option.

This option tells the compiler to find opportunities to generate separate versions of functions that take advantage of features of the specified Intel® processor.

If the compiler finds such an opportunity, it first checks whether generating a processor-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a processor-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors.

You can use more than one of the processor values by combining them. For example, you can specify `-axSSE4.1,SSSE3` (Linux and Mac OS X) or `/QaxSSE4.1,SSSE3` (Windows). You cannot combine the old style, deprecated options and the new options. For example, you cannot specify `-axSSE4.1,T` (Linux and Mac OS X) or `/QaxSSE4.1,T` (Windows).

Previous values W and K are deprecated. The details on replacements are as follows:

- Mac OS X systems: On these systems, there is no exact replacement for W or K. You can upgrade to the default option `-msse3` (IA-32 architecture) or option `-mssse3` (Intel® 64 architecture).
- Windows and Linux systems: The replacement for W is `-msse2` (Linux) or `/arch:SSE2` (Windows). There is no exact replacement for K. However, on Windows systems, `/QaxK` is interpreted as `/arch:IA32`; on Linux systems, `-axK` is interpreted as `-mia32`. You can also do one of the following:
  - Upgrade to option `-msse2` (Linux) or option `/arch:SSE2` (Windows). This will produce one code path that is specialized for Intel® SSE2. It will not run on earlier processors

- Specify the two option combination `-mia32 -axSSE2` (Linux) or `/arch:IA32 /QaxSSE2` (Windows). This combination will produce an executable that runs on any processor with IA-32 architecture but with an additional specialized Intel® SSE2 code path.

The `-ax` and `/Qax` options enable additional optimizations not enabled with option `-m` or option `/arch`.

Alternate Options

None

See Also

-
-
- `x`, `Qx`
- `m`
- `arch`

B

Specifies a directory that can be used to find include files, libraries, and executables.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-B`*`dir`*

Windows:

`None`

Arguments

dir	Is the directory to be used. If necessary, the compiler adds a directory separator character at the end of *dir*.

Default

OFF	The compiler looks for files in the directories specified in your PATH environment variable.

Description

This option specifies a directory that can be used to find include files, libraries, and executables.

The compiler uses *dir* as a prefix.

For include files, the *dir* is converted to `-I`*dir*/include. This command is added to the front of the includes passed to the preprocessor.

For libraries, the *dir* is converted to `-L`*dir*. This command is added to the front of the standard `-L` inclusions before system libraries are added.

For executables, if *dir* contains the name of a tool, such as `ld` or `as`, the compiler will use it instead of those found in the default directories.

The compiler looks for include files in *dir* /include while library files are looked for in *dir*.

Another way to get the behavior of this option is to use the environment variable `GCC_EXEC_PREFIX`.

Alternate Options

None

Bdynamic

Enables dynamic linking of libraries at run time.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-Bdynamic`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	Limited dynamic linking occurs.

Description

This option enables dynamic linking of libraries at run time. Smaller executables are created than with static linking.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bdynamic` are linked dynamically until the end of the command line or until a `-Bstatic` option is encountered. The `-Bstatic` option enables static linking of libraries.

Alternate Options

None

See Also

-
- `Bstatic`

bigobj

Increases the number of sections that an object file can contain.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
None
```

Windows:

```
/bigobj
```

Arguments

None

Default

OFF	An object file can hold up to 65,536 (2**16) addressable sections.

Description

This option increases the number of sections that an object file can contain. It increases the address capacity to 4,294,967,296(2**32).

An .obj file produced with this option can only be effectively passed to a linker that shipped in Microsoft Visual C++* 2005 or later. Linkers shipped with earlier versions of the product cannot read .obj files of this size.

This option may be helpful for .obj files that can hold more sections, such as machine generated code.

Alternate Options

None

bintext

Places a text string into the object file (.obj) being generated by the compiler.

IDE Equivalent

Windows: **Code Generation > Object Text String**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/bintext:`*`string`*

`/nobintext`

Arguments

`string`	Is the text string to go into the object file.

Default

`/nobintext`	No text string is placed in the object file.

Description

This option places a text string into the object file (.obj) being generated by the compiler. The string also gets propagated into the executable file.

For example, this option is useful if you want to place a version number or copyright information into the object and executable.

If the string contains a space or tab, the string must be enclosed by double quotation marks ("). A backslash (\) must precede any double quotation marks contained within the string.

If the command line contains multiple `/bintext` options, the last (rightmost) one is used.

Alternate Options

Linux and Mac OS X: None

Windows: `/Vstring`

Bstatic

Enables static linking of a user's library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-Bstatic`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	Default static linking occurs.

Description

This option enables static linking of a user's library.

This option is placed in the linker command line corresponding to its location on the user command line. It controls the linking behavior of any library that is passed using the command line.

All libraries on the command line following option `-Bstatic` are linked statically until the end of the command line or until a `-Bdynamic` option is encountered. The `-Bdynamic` option enables dynamic linking of libraries.

Alternate Options

None

See Also

-
- `Bdynamic`

c

Prevents linking.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-c`

Windows:

`/c`

Arguments

None

Default

OFF	Linking is performed.

Description

This option prevents linking. Compilation stops after the object file is generated.

The compiler generates an object file for each Fortran source file.

Alternate Options

Linux and Mac OS X: None

Windows: `/compile-only`, `/nolink`

C

See check.

CB

See check.

ccdefault

Specifies the type of carriage control used when a file is displayed at a terminal screen.

IDE Equivalent

Windows: **Run-time > Default Output Carriage Control**

Linux: None

Mac OS X: **Run-time > Default Output Carriage Control**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ccdefault` *`keyword`*

Windows:

`/ccdefault:`*`keyword`*

Arguments

keyword — Specifies the carriage-control setting to use. Possible values are:

`none`	Tells the compiler to use no carriage control processing.
`default`	Tells the compiler to use the default carriage-control setting.
`fortran`	Tells the compiler to use normal Fortran interpretation of the first character. For example, the character 0 causes output of a blank line before a record.
`list`	Tells the compiler to output one line feed between records.

Default

`ccdefault default` — The compiler uses the default carriage control setting.

Description

This option specifies the type of carriage control used when a file is displayed at a terminal screen (units 6 and *). It provides the same functionality as using the CARRIAGECONTROL specifier in an OPEN statement.

The default carriage-control setting can be affected by the `vms` option. If `vms` is specified with `ccdefault default`, carriage control defaults to normal Fortran interpretation (`ccdefault fortran`) if the file is formatted and the unit is connected to a terminal. If `novms` (the default) is specified with `ccdefault default`, carriage control defaults to list (`ccdefault list`).

Alternate Options

None

check

Checks for certain conditions at run time.

IDE Equivalent

Windows:

Run-time > Runtime Error Checking (`/nocheck`, `/check:all`, or `/xcheck:none`)

Run-time > Check Array and String Bounds (`/check:[no]bounds`)

Run-time > Check Uninitialized Variables (`/check:[no]uninit`)

Run-time > Check Edit Descriptor Data Type (`/check:[no]format`)

Run-time > Check Edit Descriptor Data Size (`/check:[no]output_conversion`)

Run-time > Check For Actual Arguments Using Temporary Storage (`/check:[no]arg_temp_created`)

Run-time > **Check For Null Pointers and Allocatable Array References** (`/check:[no]pointers`)

Linux: None

Mac OS X: Run-time > Runtime Error Checking (`-check all, -check none`)

Run-time > Check Array and String Bounds (`-check [no]bounds`)

Run-time > Check Edit Descriptor Data Type (`-check [no]format`)

Run-time > Check Edit Descriptor Data Size (`-check [no]output_conversion`)

Run-time > Check For Actual Arguments Using Temporary Storage (`-check [no]arg_temp_created`)

Run-time > Check for Uninitialized Variables (`-check [no]uninit`)

Run-time > Check For Null Pointers and Allocatable Array References (`/check:[no]pointers`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-check [keyword]`

`-nocheck`

Windows:

`/check[:keyword]`

`/nocheck`

Arguments

keyword Specifies the conditions to check. Possible values are:

`none`	Disables all check options.
`[no]arg_temp_created`	Determines whether checking occurs for actual arguments before routine calls.
`[no]bounds`	Determines whether checking occurs for array subscript and character substring expressions.
`[no]format`	Determines whether checking occurs for the data type of an item being formatted for output.
`[no]output_conversion`	Determines whether checking occurs for the fit of data items within a designated format descriptor field.
`[no]pointers`	Determines whether checking occurs for certain disassociated or uninitialized pointers or unallocated allocatable objects.
`[no]uninit`	Determines whether checking occurs for uninitialized variables.
`all`	Enables all check options.

Default

`nocheck` No checking is performed for run-time failures. Note that if option `vms` is specified, the defaults are `check format` and `check output_conversion`.

Description

This option checks for certain conditions at run time.

Option	Description
`check none`	Disables all check options (same as `nocheck`).

Option	Description
`check arg_temp_created`	Enables run-time checking on whether actual arguments are copied into temporary storage before routine calls. If a copy is made at run-time, an informative message is displayed.
`check bounds`	Enables compile-time and run-time checking for array subscript and character substring expressions. An error is reported if the expression is outside the dimension of the array or the length of the string. For array bounds, each individual dimension is checked. Array bounds checking is not performed for arrays that are dummy arguments in which the last dimension bound is specified as * or when both upper and lower dimensions are 1. Once the program is debugged, omit this option to reduce executable program size and slightly improve run-time performance.
`check format`	Issues the run-time FORVARMIS fatal error when the data type of an item being formatted for output does not match the format descriptor being used (for example, a REAL*4 item formatted with an I edit descriptor). With check noformat, the data item is formatted using the specified descriptor unless the length of the item cannot accommodate the descriptor (for example, it is still an error to pass an INTEGER*2 item to an E edit descriptor).
`check output_conversion`	Issues the run-time OUTCONERR continuable error message when a data item is too large to fit in a designated format descriptor field without loss of significant digits. Format truncation occurs, the field is filled with asterisks (*), and execution continues.
`check pointers`	Enables run-time checking for disassociated or uninitialized Fortran pointers, unallocated allocatable objects, and integer pointers that are uninitialized.
`check uninit`	Enables run-time checking for uninitialized variables. If a variable is read before it is written, a run-time error routine will be called. Only local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, and LOGICAL without the SAVE attribute are checked.

Option	Description
`check all`	Enables all check options. This is the same as specifying `check` with no keyword.

To get more detailed location information about where an error occurred, use option `traceback`.

Alternate Options

`check none`	Linux and Mac OS X: `-nocheck` Windows: `/nocheck`, `/4Nb`
`check bounds`	Linux and Mac OS X: `-CB` Windows: `/CB`
`check uninit`	Linux and Mac OS X: `-CU` Windows: `/RTCu`, `/CU`
`check all`	Linux and Mac OS X: `-check, -C` Windows: `/check`, `/4Yb`, `/C`

See Also

-
- `traceback`

cm

See warn.

common-args, Qcommon-args

See assume.

compile-only

See c.

complex-limited-range, Qcomplex-limited-range

Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

IDE Equivalent

Windows: **Floating point > Limit COMPLEX Range**

Linux: None

Mac OS X: **Floating point > Limit COMPLEX Range**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-complex-limited-range
-no-complex-limited-range
```

Windows:

```
/Qcomplex-limited-range
/Qcomplex-limited-range-
```

Arguments

None

Default

`-no-complex-limited-range` or `/Qcomplex-limited-range-`	Basic algebraic expansions of some arithmetic operations involving data of type COMPLEX are disabled.

Description

This option determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

When the option is enabled, this can cause performance improvements in programs that use a lot of COMPLEX arithmetic. However, values at the extremes of the exponent range may not compute correctly.

Alternate Options

None

convert

Specifies the format of unformatted files containing numeric data.

IDE Equivalent

Windows: **Compatibility > Unformatted File Conversion**

Linux: None

Mac OS X: **Compatibility > Unformatted File Conversion**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-convert` *keyword*

Windows:

`/convert:`*keyword*

Arguments

keyword

Specifies the format for the unformatted numeric data. Possible values are:

`native`	Specifies that unformatted data should not be converted.
`big_endian`	Specifies that the format will be big endian for integer data and big endian IEEE floating-point for real and complex data.
`cray`	Specifies that the format will be big endian for integer data and CRAY* floating-point for real and complex data.
`fdx` (Linux, Mac OS X)	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, D_floating, and X_floating for real and complex data.
`fgx` (Linux, Mac OS X)	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, G_floating, and X_floating for real and complex data.
`ibm`	Specifies that the format will be big endian for integer data and IBM* System\370 floating-point format for real and complex data.
`little_endian`	Specifies that the format will be little endian for integer data and little endian IEEE floating-point for real and complex data.
`vaxd`	Specifies that the format will be little endian for integer data, and VAX* processor floating-point format F_floating, D_floating, and H_floating for real and complex data.

`vaxg`	Specifies that the format will be little endian for integer data, and VAX processor floating-point format F_floating, G_floating, and H_floating for real and complex data.

Default

`convert native`	No conversion is performed on unformatted files containing numeric data.

Description

This option specifies the format of unformatted files containing numeric data.

Option	Description
`convert native`	Specifies that unformatted data should not be converted.
`convert big_endian`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and big endian IEEE floating-point for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32.
`convert cray`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and CRAY* floating-point for REAL*8 or COMPLEX*16.
`convert fdx`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and X_floating for REAL*16 or COMPLEX*32.
`convert fgx`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, G_floating for REAL*8 or COMPLEX*16, and X_floating for REAL*16 or COMPLEX*32.
`convert ibm`	Specifies that the format will be big endian for INTEGER*1, INTEGER*2, or INTEGER*4, and IBM* System\370 floating-point format for REAL*4 or COMPLEX*8 (IBM short 4) and REAL*8 or COMPLEX*16 (IBM long 8).

Option	Description
`convert little_endian`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8 and little endian IEEE floating-point for REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16, or COMPLEX*32.
`convert vaxd`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, D_floating for REAL*8 or COMPLEX*16, and H_floating for REAL*16 or COMPLEX*32.
`convert vaxg`	Specifies that the format will be little endian for INTEGER*1, INTEGER*2, INTEGER*4, or INTEGER*8, and VAX processor floating-point format F_floating for REAL*4 or COMPLEX*8, G_floating for REAL*8 or COMPLEX*16, and H_floating for REAL*16 or COMPLEX*32.

Alternate Options

None

cpp, Qcpp

See fpp, Qfpp.

CU

See check.

cxxlib

Determines whether the compile links using the C++ run-time libraries provided by gcc.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-cxxlib[=dir]`

`-cxxlib-nostd`

`-no-cxxlib`

Windows:

`None`

Arguments

dir	Is an optional top-level location for the gcc binaries and libraries.

Default

`-no-cxxlib`	The compiler uses the default run-time libraries and does not link to any additional C++ run-time libraries.

Description

This option determines whether the compile links using the C++ run-time libraries provided by gcc.

Option `-cxxlib-nostd` prevents the compiler from linking with the standard C++ library. It is only useful for mixed-language applications.

Alternate Options

None

See Also

-

Building Applications: Options for Interoperability

D

Defines a symbol name that can be associated with an optional value.

IDE Equivalent

Windows: **General > Preprocessor Definitions**

Preprocessor> Preprocessor Definitions

Preprocessor > Preprocessor Definitions to FPP only

Linux: None

Mac OS X: **Preprocessor > Preprocessor Definitions**

Preprocessor > Preprocessor Definitions to FPP only

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Dname[=value]`

Windows:

`/Dname[=value]`

Arguments

name	Is the name of the symbol.
value	Is an optional integer or an optional character string delimited by double quotes; for example, `Dname=string`.

Default

`noD`	Only default symbols or macros are defined.

Description

Defines a symbol name that can be associated with an optional value.
This definition is used during preprocessing.

If a *value* is not specified, *name* is defined as "1".

If you want to specify more than one definition, you must use separate D options.

If you specify noD, all preprocessor definitions apply only to fpp and not to Intel® Fortran conditional compilation directives. To use this option, you must also specify option fpp.

CAUTION. On Linux and Mac OS X systems, if you are not specifying a *value*, do not use D for *name*, because it will conflict with the –DD option.

Alternate Options

D	Linux and Mac OS X: None Windows: /define:*name[=value]*
noD	Linux and Mac OS X: -nodefine Windows: /nodefine

See Also

•

Building Applications: Predefined Preprocessor Symbols

d-lines, Qd-lines

Compiles debug statements.

IDE Equivalent

Windows: **Language > Compile Lines With D in Column 1**

Linux: None

Mac OS X: **Language > Compile Lines With D in Column 1**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-d-lines`

`-nod-lines`

Windows:

`/d-lines`

`/nod-lines`

`/Qd-lines`

Arguments

None

Default

`nod-lines`	Debug lines are treated as comment lines.

Description

This option compiles debug statements. It specifies that lines in fixed-format files that contain a D in column 1 (debug statements) should be treated as source code.

Alternate Options

Linux and Mac OS X: `-DD`

Windows: None

dbglibs

Tells the linker to search for unresolved references in a debug run-time library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/dbglibs`

`/nodbglibs`

Arguments

None

Default

`/nodbglibs`	The linker does not search for unresolved references in a debug run-time library.

Description

This option tells the linker to search for unresolved references in a debug run-time library.

The following table shows which options to specify for a debug run-time library:

Type of Library	Options Required	Alternate Option
Debug single-threaded	`/libs:static` `/dbglibs`	`/MLd` (this is a deprecated option)

Debug multithreaded	`/libs:static`
	`/threads`
	`/dbglibs`
		`/MTd`

Multithreaded debug DLLs	`/libs:dll`
	`/threads`
	`/dbglibs`
		`/MDd`

Debug Fortran QuickWin multi-thread applications	`/libs:qwin` `/dbglibs`	None

Type of Library	Options Required	Alternate Option
Debug Fortran standard graphics (QuickWin single-thread) applications	`/libs:qwins` `/dbglibs`	None

Alternate Options

None

See Also

- Building Applications: Specifying Consistent Library Types; Programming with Mixed Languages Overview

DD

See dlines, Qdlines.

debug (Linux* OS and Mac OS* X)

Enables or disables generation of debugging information.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-debug[keyword]`

Windows:

`None`

Arguments

keyword Is the type of debugging information to be generated. Possible values are:

`none`	Disables generation of debugging information.
`full` or `all`	Generates complete debugging information.
`minimal`	Generates line number information for debugging.
`[no]emit_column`	Determines whether the compiler generates column number information for debugging.
`[no]inline-debug-info`	Determines whether the compiler generates enhanced debug information for inlined code.
`[no]semantic-stepping`	Determines whether the compiler generates enhanced debug information useful for breakpoints and stepping.
`[no]variable-locations`	Determines whether the compiler generates enhanced debug information useful in finding scalar local variables.
`extended`	Sets keyword values `semantic-stepping` and `variable-locations`.

For information on the non-default settings for these keywords, see the Description section.

Default

`-debug none` No debugging information is generated.

Description

This option enables or disables generation of debugging information.

Note that if you turn debugging on, optimization is turned off.

Keywords `semantic-stepping`, `inline-debug-info`, `variable-locations`, and `extended` can be used in combination with each other. If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
`-debug none`	Disables generation of debugging information.
`-debug full` or `-debug all`	Generates complete debugging information. It is the same as specifying `-debug` with no keyword.
`-debug minimal`	Generates line number information for debugging.
`-debug emit_column`	Generates column number information for debugging.
`-debug inline-debug-info`	Generates enhanced debug information for inlined code. It provides more information to debuggers for function call traceback.
`-debug semantic-stepping`	Generates enhanced debug information useful for breakpoints and stepping. It tells the debugger to stop only at machine instructions that achieve the final effect of a source statement. For example, in the case of an assignment statement, this might be a store instruction that assigns a value to a program variable; for a function call, it might be the machine instruction that executes the call. Other instructions generated for those source statements are not displayed during stepping. This option has no impact unless optimizations have also been enabled.
`-debug variable-locations`	Generates enhanced debug information useful in finding scalar local variables. It uses a feature of the Dwarf object module known as "location lists". This feature allows the run-time locations of local scalar variables to be specified more accurately; that is, whether, at a given position in the code, a variable value is found in memory or a machine register.

Option	Description
`-debug extended`	Sets keyword values `semantic-stepping` and `variable-locations`. It also tells the compiler to include column numbers in the line information.

On Linux* systems, debuggers read debug information from executable images. As a result, information is written to object files and then added to the executable by the linker. On Mac OS* X systems, debuggers read debug information from object files. As a result, the executables don't contain any debug information. Therefore, if you want to be able to debug on these systems, you must retain the object files.

Alternate Options

For `debug full`, `-debug all`, or `-debug`	Linux and Mac OS X: `-g` Windows: `/debug:full`, `/debug:all`, or `/debug`
For `-debug inline-debug-info`	Linux and Mac OS X: `-inline-debug-info` (this is a deprecated option) Windows: None

See Also

-
- `debug` (Windows*)

Building Applications: Debugging Overview

debug (Windows* OS)

Enables or disables generation of debugging information.

IDE Equivalent

Windows: **General > Debug Information Format** (`/debug:minimal`, `/debug:full`)

Debug > Enable Parallel Debug Checks (`/debug:parallel`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/debug[:keyword]`

`/nodebug`

Arguments

keyword	Is the type of debugging information to be generated. Possible values are:	
	`none`	Generates no symbol table information.
	`full` or `all`	Generates complete debugging information.
	`minimal`	Generates line numbers and minimal debugging information.
	`partial`	Deprecated. Generates global symbol table information needed for linking.
	`[no]parallel` (i32, i64em)	Determines whether the compiler generates parallel debug code instrumentations useful for thread data sharing and reentrant call detection. For this setting to be effective, option `/Qopenmp` must be set.

For information on the non-default settings for these keywords, see the Description section.

Default

`/debug:none`	This is the default on the command line and for a release configuration in the IDE.
`/debug:full`	This is the default for a debug configuration in the IDE.

Description

This option enables or disables generation of debugging information. It is passed to the linker.

Note that if you turn debugging on, optimization is turned off.

If conflicting keywords are used in combination, the last one specified on the command line has precedence.

Option	Description
`/debug:none`	Disables generation of debugging information. It is the same as specifying /nodebug.
`/debug:full` or `/debug:all`	Generates complete debugging information. It produces symbol table information needed for full symbolic debugging of unoptimized code and global symbol information needed for linking. It is the same as specifying `/debug` with no keyword. If you specify `/debug:full` for an application that makes calls to C library routines and you need to debug calls into the C library, you should also specify `/dbglibs` to request that the appropriate C debug library be linked against.
`/debug:minimal`	Generates line number information for debugging. It produces global symbol information needed for linking, but not local symbol table information needed for debugging.
`/debug:partial`	Generates global symbol table information needed for linking, but not local symbol table information needed for debugging. This option is deprecated and is not available in the IDE.
`/debug:parallel`	Generates parallel debug code instrumentations needed for the thread data sharing and reentrant call detection of the Intel® Parallel Debugger Extension. This option is only available on IA-32 and Intel® 64 architectures.

Alternate Options

For `/debug:minimal`	Linux and Mac OS X: None Windows: `/Zd` (this is a deprecated option)
For `/debug:full` or `/debug`	Linux and Mac OS X: None Windows: `/Zi`, `/Z7`

See Also

-
- `dbglibs`
- `debug` (Linux* and Mac OS* X)
- Debugging Fortran Programs

debug-parameters

Tells the compiler to generate debug information for PARAMETERs used in a program.

IDE Equivalent

Windows: **Debugging > Information for PARAMETER Constants**

Linux: None

Mac OS X: **Debug > Information for PARAMETER Constants**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-debug-parameters [keyword]`

`-nodebug-parameters`

Windows:

`/debug-parameters[:keyword]`

`/nodebug-parameters`

Arguments

keyword — Are the PARAMETERs to generate debug information for. Possible values are:

- `none` — Generates no debug information for any PARAMETERs used in the program. This is the same as specifying `nodebug-parameters`.
- `used` — Generates debug information for only PARAMETERs that have actually been referenced in the program. This is the default if you do not specify a *keyword*.
- `all` — Generates debug information for all PARAMETERs defined in the program.

Default

`nodebug-parameters` — The compiler generates no debug information for any PARAMETERs used in the program. This is the same as specifying *keyword* `none`.

Description

This option tells the compiler to generate debug information for PARAMETERs used in a program.

Note that if a .mod file contains PARAMETERs, debug information is only generated for the PARAMETERs that have actually been referenced in the program, even if you specify *keyword* `all`.

Alternate Options

None

define

See D.

diag, Qdiag

Controls the display of diagnostic information.

IDE Equivalent

Windows: **Diagnostics > Disable Specific Diagnostics** (`/Qdiag-disable:id`)

Diagnostics > Level of Source Code Analysis (`/Qdiag-enable[:sc1,sc2,sc3]`)

Linux: None

Mac OS X: **Diagnostics > Disable Specific Diagnostics** (`-diag-disable id`)

Diagnostics > Level of Source Code Analysis (`-diag-enable [sc1,sc2,sc3]`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-`*`type diag-list`*

Windows:

`/Qdiag-`*`type:diag-list`*

Arguments

`type`	Is an action to perform on diagnostics. Possible values are:	
	`enable`	Enables a diagnostic message or a group of messages.
	`disable`	Disables a diagnostic message or a group of messages.
	`error`	Tells the compiler to change diagnostics to errors.
	`warning`	Tells the compiler to change diagnostics to warnings.
	`remark`	Tells the compiler to change diagnostics to remarks (comments).
`diag-list`	Is a diagnostic group or ID value. Possible values are:	
	`driver`	Specifies diagnostic messages issued by the compiler driver.
	`vec`	Specifies diagnostic messages issued by the vectorizer.
	`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).

`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
`sc[n]`	Specifies diagnostic messages issued by the Source Checker. *n* can be any of the following: 1, 2, 3. For more details on these values, see below. This value is equivalent to deprecated value sv[n].
`warn`	Specifies diagnostic messages that have a "warning" severity level.
`error`	Specifies diagnostic messages that have an "error" severity level.
`remark`	Specifies diagnostic messages that are remarks or comments.
`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. This diagnostic group is only available on IA-32 architecture and Intel® 64 architecture.
`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each id.
`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each tag.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information. Diagnostic messages are output to stderr unless compiler option `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) is specified.

When *diag-list* value "warn" is used with the Source Checker (sc) diagnostics, the following behavior occurs:

- Option `-diag-enable warn` (Linux and Mac OS X) and `/Qdiag-enable:warn` (Windows) enable all Source Checker diagnostics except those that have an "error" severity level. They enable all Source Checker warnings, cautions, and remarks.
- Option `-diag-disable warn` (Linux and Mac OS X) and `/Qdiag-disable:warn` (Windows) disable all Source Checker diagnostics except those that have an "error" severity level. They suppress all Source Checker warnings, cautions, and remarks.

The following table shows more information on values you can specify for diag-list item sc.

***diag-list* Item**	**Description**	
`sc[n]`	The value of *n* for Source Checker messages can be any of the following:	
	1	Produces the diagnostics with severity level set to all critical errors.
	2	Produces the diagnostics with severity level set to all errors. This is the default if n is not specified.
	3	Produces the diagnostics with severity level set to all errors and warnings.

To control the diagnostic information reported by the vectorizer, use the `-vec-report` (Linux and Mac OS X) or `/Qvec-report` (Windows) option.

To control the diagnostic information reported by the auto-parallelizer, use the `-par-report` (Linux and Mac OS X) or `/Qpar-report` (Windows) option.

Alternate Options

enable vec	Linux and Mac OS X: `-vec-report` Windows: `/Qvec-report`
disable vec	Linux and Mac OS X: `-vec-report0` Windows: `/Qvec-report0`

enable par	Linux and Mac OS X: `-par-report` Windows: `/Qpar-report`
disable par	Linux and Mac OS X: `-par-report0` Windows: `/Qpar-report0`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable 117,230,450     ! Linux and Mac OS X systems

/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable vec -diag-warning vec      ! Linux and Mac OS X systems

/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable par     ! Linux and Mac OS X systems

/Qdiag-disable:par    ! Windows systems
```

The following example shows how to produce Source Checker diagnostic messages for all critical errors:

```
-diag-enable sc1      ! Linux and Mac OS X systems

/Qdiag-enable:sc1     ! Windows system
```

The following example shows how to cause Source Checker diagnostics (and default diagnostics) to be sent to a file:

```
-diag-enable sc -diag-file=stat_ver_msg   ! Linux and Mac OS X systems


/Qdiag-enable:sc /Qdiag-file:stat_ver_msg ! Windows systems
```

Note that you need to enable the Source Checker diagnostics before you can send them to a file. In this case, the diagnostics are sent to file stat_ver_msg.diag. If a file name is not specified, the diagnostics are sent to name-of-the-first-source-file.diag.

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error warn,remark    ! Linux and Mac OS X systems

/Qdiag-error:warn,remark   ! Windows systems
```

See Also

-
-
-
-
- diag-dump, Qdiag-dump
- diag-id-numbers, Qdiag-id-numbers
- diag-file, Qdiag-file
- par-report, Qpar-report
- vec-report, Qvec-report

diag-dump, Qdiag-dump

Tells the compiler to print all enabled diagnostic messages and stop compilation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-diag-dump
```

Windows:

```
/Qdiag-dump
```

Arguments

None

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to print all enabled diagnostic messages and stop compilation. The diagnostic messages are output to `stdout`.

This option prints the enabled diagnostics from all possible diagnostics that the compiler can issue, including any default diagnostics.

If `-diag-enable` *`diag-list`* (Linux and Mac OS X) or `/Qdiag-enable` *`diag-list`* (Windows) is specified, the print out will include the *`diag-list`* diagnostics.

Alternate Options

None

Example

The following example adds vectorizer diagnostic messages to the printout of default diagnostics:

```
-diag-enable vec -diag-dump         ! Linux and Mac OS X systems
/Qdiag-enable:vec /Qdiag-dump    ! Windows systems
```

See Also

-
-
- `diag, Qdiag`

diag, Qdiag

Controls the display of diagnostic information.

IDE Equivalent

Windows: **Diagnostics > Disable Specific Diagnostics** (`/Qdiag-disable:id`)

Diagnostics > Level of Source Code Analysis (`/Qdiag-enable[:sc1,sc2,sc3]`)

Linux: None

Mac OS X: **Diagnostics > Disable Specific Diagnostics** (`-diag-disable id`)

Diagnostics > Level of Source Code Analysis (`-diag-enable [sc1,sc2,sc3]`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-`*`type diag-list`*

Windows:

`/Qdiag-`*`type:diag-list`*

Arguments

type	Is an action to perform on diagnostics. Possible values are:	
	`enable`	Enables a diagnostic message or a group of messages.
	`disable`	Disables a diagnostic message or a group of messages.
	`error`	Tells the compiler to change diagnostics to errors.
	`warning`	Tells the compiler to change diagnostics to warnings.
	`remark`	Tells the compiler to change diagnostics to remarks (comments).
diag-list	Is a diagnostic group or ID value. Possible values are:	
	`driver`	Specifies diagnostic messages issued by the compiler driver.
	`vec`	Specifies diagnostic messages issued by the vectorizer.
	`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).
	`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
	`sc[n]`	Specifies diagnostic messages issued by the Source Checker. *n* can be any of the following: 1, 2, 3. For more details on these values, see below. This value is equivalent to deprecated value sv[n].

`warn`	Specifies diagnostic messages that have a "warning" severity level.
`error`	Specifies diagnostic messages that have an "error" severity level.
`remark`	Specifies diagnostic messages that are remarks or comments.
`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. This diagnostic group is only available on IA-32 architecture and Intel® 64 architecture.
`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each id.
`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each tag.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information. Diagnostic messages are output to stderr unless compiler option `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) is specified.

When *`diag-list`* value "warn" is used with the Source Checker (sc) diagnostics, the following behavior occurs:

- Option `-diag-enable warn` (Linux and Mac OS X) and `/Qdiag-enable:warn` (Windows) enable all Source Checker diagnostics except those that have an "error" severity level. They enable all Source Checker warnings, cautions, and remarks.

- Option `-diag-disable warn` (Linux and Mac OS X) and `/Qdiag-disable:warn` (Windows) disable all Source Checker diagnostics except those that have an "error" severity level. They suppress all Source Checker warnings, cautions, and remarks.

The following table shows more information on values you can specify for diag-list item sc.

diag-list Item	Description	
`sc[n]`	The value of *n* for Source Checker messages can be any of the following:	
	1	Produces the diagnostics with severity level set to all critical errors.
	2	Produces the diagnostics with severity level set to all errors. This is the default if n is not specified.
	3	Produces the diagnostics with severity level set to all errors and warnings.

To control the diagnostic information reported by the vectorizer, use the `-vec-report` (Linux and Mac OS X) or `/Qvec-report` (Windows) option.

To control the diagnostic information reported by the auto-parallelizer, use the `-par-report` (Linux and Mac OS X) or `/Qpar-report` (Windows) option.

Alternate Options

enable vec	Linux and Mac OS X: `-vec-report` Windows: `/Qvec-report`
disable vec	Linux and Mac OS X: `-vec-report0` Windows: `/Qvec-report0`
enable par	Linux and Mac OS X: `-par-report` Windows: `/Qpar-report`
disable par	Linux and Mac OS X: `-par-report0` Windows: `/Qpar-report0`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable 117,230,450     ! Linux and Mac OS X systems
/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable vec -diag-warning vec      ! Linux and Mac OS X systems

/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable par     ! Linux and Mac OS X systems

/Qdiag-disable:par    ! Windows systems
```

The following example shows how to produce Source Checker diagnostic messages for all critical errors:

```
-diag-enable sc1      ! Linux and Mac OS X systems

/Qdiag-enable:sc1     ! Windows system
```

The following example shows how to cause Source Checker diagnostics (and default diagnostics) to be sent to a file:

```
-diag-enable sc -diag-file=stat_ver_msg   ! Linux and Mac OS X systems


/Qdiag-enable:sc /Qdiag-file:stat_ver_msg ! Windows systems
```

Note that you need to enable the Source Checker diagnostics before you can send them to a file. In this case, the diagnostics are sent to file stat_ver_msg.diag. If a file name is not specified, the diagnostics are sent to name-of-the-first-source-file.diag.

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error warn,remark    ! Linux and Mac OS X systems

/Qdiag-error:warn,remark   ! Windows systems
```

See Also

-
-
-
-
- diag-dump, Qdiag-dump
- diag-id-numbers, Qdiag-id-numbers
- diag-file, Qdiag-file
- par-report, Qpar-report

- vec-report, Qvec-report

diag-enable sc-include, Qdiag-enable:sc-include

Tells a source code analyzer to process include files and source files when issuing diagnostic messages.

IDE Equivalent

Windows: **Diagnostics > Analyze Include Files**

Linux: None

Mac OS X: **Diagnostics > Analyze Include Files**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-diag-enable sc-include
```

Windows:

```
/Qdiag-enable:sc-include
```

Arguments

None

Default

OFF	The compiler issues certain diagnostic messages by default. If the Source Checker is enabled, include files are not analyzed by default.

Description

This option tells a source code analyzer (Source Checker) to process include files and source files when issuing diagnostic messages. Normally, when Source Checker diagnostics are enabled, only source files are analyzed.

To use this option, you must also specify `-diag-enable sc` (Linux and Mac OS X) or `/Qdiag-enable:sc` (Windows) to enable the Source Checker diagnostics, or `-diag-enable sc-parallel` (Linux and Mac OS X) or `/Qdiag-enable:sc-parallel` (Windows) to enable parallel lint.

Alternate Options

Linux and Mac OS X: `-diag-enable sv-include` (this is a deprecated option)

Windows: `/Qdiag-enable:sv-include` (this is a deprecated option)

Example

The following example shows how to cause include files to be analyzed as well as source files:

```
-diag-enable sc -diag-enable sc-include    ! Linux and Mac OS systems

/Qdiag-enable:sc /Qdiag-enable:sc-include  ! Windows systems
```

In the above example, the first compiler option enables Source Checker messages. The second compiler option causes include files referred to by the source file to be analyzed also.

See Also

-
-
- `diag-enable sc-parallel`, `Qdiag-enable:sc-parallel`
- `diag`, `Qdiag`

diag-enable sc-parallel, Qdiag-enable:sc-parallel

Enables analysis of parallelization in source code (parallel lint diagnostics).

IDE Equivalent

Windows: **Diagnostics > Level of Source Code Parallelization Analysis**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-diag-enable sc-parallel[n]`

Windows:

`/Qdiag-enable:sc-parallel[n]`

Arguments

n	Is the level of analysis to perform. Possible values are:
1	Produces the diagnostics with severity level set to all critical errors.
2	Tells the compiler to generate a report with the medium level of detail. Produces the diagnostics with severity level set to all errors. This is the default if *n* is not specified.
3	Produces the diagnostics with severity level set to all errors and warnings.

Default

OFF	The compiler does not analyze parallelization in source code.

Description

This option enables analysis of parallelization in source code (parallel lint diagnostics). Currently, this analysis uses OpenMP directives, so this option has no effect unless option `/Qopenmp` (Windows) or option `-openmp` (Linux and Mac OS X) is set.

Parallel lint performs interprocedural source code analysis to identify mistakes when using parallel directives. It reports various problems that are difficult to find, including data dependency and potential deadlocks.

Source Checker diagnostics (enabled by `/Qdiag-enable:sc` on Windows* OS or `-diag-enable sc` on Linux* OS and Mac OS* X) are a superset of parallel lint diagnostics. Therefore, if Source Checker diagnostics are enabled, the parallel lint option is not taken into account.

Alternate Options

None

See Also

-
-
- diag, Qdiag

diag-error-limit, Qdiag-error-limit

Specifies the maximum number of errors allowed before compilation stops.

IDE Equivalent

Windows: **Compilation Diagnostics > Error Limit**

Linux: None

Mac OS X: **Compiler Diagnostics > Error Limit**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-error-limit`*n*

`-no-diag-error-limit`

Windows:

`/Qdiag-error-limit:`*n*

`/Qdiag-error-limit-`

Arguments

n	Is the maximum number of error-level or fatal-level compiler errors allowed.

Default

30	A maximum of 30 error-level and fatal-level messages are allowed.

Description

This option specifies the maximum number of errors allowed before compilation stops. It indicates the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line.

If you specify `-no-diag-error-limit` (Linux and Mac OS X) or `/Qdiag-error-limit-` (Windows) on the command line, there is no limit on the number of errors that are allowed.

If the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

Alternate Options

Linux and Mac OS X: `-error-limit` and `-noerror-limit`

Windows: `/error-limit` and `/noerror-limit`

diag-file, Qdiag-file

Causes the results of diagnostic analysis to be output to a file.

IDE Equivalent

Windows: **Diagnostics > Diagnostics File**

Linux: None

Mac OS X: **Diagnostics > Diagnostics File**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-file[=`*`file`*`]`

Windows:

`/Qdiag-file[:`*`file`*`]`

Arguments

file	Is the name of the file for output.

Default

OFF	Diagnostic messages are output to stderr.

Description

This option causes the results of diagnostic analysis to be output to a file. The file is placed in the current working directory.

If *file* is specified, the name of the file is *file.diag*. The file can include a file extension; for example, if *file.ext* is specified, the name of the file is *file.ext*.

If *file* is not specified, the name of the file is *name-of-the-first-source-file.diag*. This is also the name of the file if the name specified for file conflicts with a source file name provided in the command line.

NOTE. If you specify `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) and you also specify `-diag-file-append` (Linux and Mac OS X) or `/Qdiag-file-append` (Windows), the last option specified on the command line takes precedence.

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be output to a file named `my_diagnostics.diag`:

```
-diag-file=my_diagnostics       ! Linux and Mac OS X systems
/Qdiag-file:my_diagnostics      ! Windows systems
```

See Also

-
-
- diag, Qdiag
- diag-file-append, Qdiag-file-append

diag-file-append, Qdiag-file-append

Causes the results of diagnostic analysis to be appended to a file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-file-append[=file]`

Windows:

`/Qdiag-file-append[:file]`

Arguments

file	Is the name of the file to be appended to. It can include a path.

Default

OFF	Diagnostic messages are output to `stderr`.

Description

This option causes the results of diagnostic analysis to be appended to a file. If you do not specify a path, the driver will look for *file* in the current working directory.

If *file* is not found, then a new file with that name is created in the current working directory. If the name specified for file conflicts with a source file name provided in the command line. the name of the file is *name-of-the-first-source-file.diag*.

NOTE. If you specify `-diag-file-append` (Linux and Mac OS X) or `/Qdiag-file-append` (Windows) and you also specify `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows), the last option specified on the command line takes precedence.

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be appended to a file named my_diagnostics.txt:

```
-diag-file-append=my_diagnostics.txt       ! Linux and Mac OS X systems
/Qdiag-file-append:my_diagnostics.txt      ! Windows systems
```

See Also

-
-
- diag, Qdiag
- diag-file, Qdiag-file

diag-id-numbers, Qdiag-id-numbers

Determines whether the compiler displays diagnostic messages by using their ID number values.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-diag-id-numbers
-no-diag-id-numbers
```

Windows:

```
/Qdiag-id-numbers
/Qdiag-id-numbers-
```

Arguments

None

Default

`-diag-id-numbers` or `/Qdiag-id-numbers`	The compiler displays diagnostic messages by using their ID number values.

Description

This option determines whether the compiler displays diagnostic messages by using their ID number values. If you specify `-no-diag-id-numbers` (Linux and Mac OS X) or `/Qdiag-id-numbers-` (Windows), mnemonic names are output for driver diagnostics only.

Alternate Options

None

See Also

-
-
- `diag`, `Qdiag`

diag-once, Qdiag-once

Tells the compiler to issue one or more diagnostic messages only once.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-once`*`id`*`[,`*`id`*`,...]`

Windows:

`/Qdiag-once:id[,id,...]`

Arguments

id	Is the ID number of the diagnostic message. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each *id*.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to issue one or more diagnostic messages only once.

Alternate Options

None

dll

Specifies that a program should be linked as a dynamic-link (DLL) library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/dll`

Arguments

None

Default

OFF	The program is not linked as a dynamic-link (DLL) library.

Description

This option specifies that a program should be linked as a dynamic-link (DLL) library instead of an executable (.exe) file. It overrides any previous specification of run-time routines to be used and enables the `/libs:dll` option.

If you use this option with the `/libs:qwin` or `/libs:qwins` option, the compiler issues a warning.

Alternate Options

Linux and Mac OS X: None

Windows: `/LD`

double-size

Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX variables.

IDE Equivalent

Windows: **Data > Default Double Precision KIND**

Linux: None

Mac OS X: **Data > Default Double Precision KIND**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-double-size` *size*

Windows:

`/double-size:`*`size`*

Arguments

`size`	Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics. Possible values are: 64 (KIND=8) or 128 (KIND=16).

Default

64	DOUBLE PRECISION variables are defined as REAL*8 and DOUBLE COMPLEX variables are defined as COMPLEX*16.

Description

This option defines the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX declarations, constants, functions, and intrinsics.

Option	Description
`double-size 64`	Defines DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL(KIND=8) (REAL*8) and defines DOUBLE COMPLEX declarations, functions, and intrinsics as COMPLEX(KIND=8) (COMPLEX*16).
`double-size 128`	Defines DOUBLE PRECISION declarations, constants, functions, and intrinsics as REAL(KIND=16) (REAL*16) and defines DOUBLE COMPLEX declarations, functions, and intrinsics as COMPLEX(KIND=16) (COMPLEX*32).

Alternate Options

None

dps, Qdps

See altparam.

dryrun

Specifies that driver tool commands should be shown but not executed.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-dryrun`

Windows:

`None`

Arguments

None

Default

OFF	No tool commands are shown, but they are executed.

Description

This option specifies that driver tool commands should be shown but not executed.

Alternate Options

None

See Also

-
- v

dumpmachine

Displays the target machine and operating system configuration.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-dumpmachine`

Windows:

`None`

Arguments

None

Default

OFF	The compiler does not display target machine or operating system information.

Description

This option displays the target machine and operating system configuration. No compilation is performed.

Alternate Options

None

dynamic-linker

Specifies a dynamic linker other than the default.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-dynamic-linker` *`file`*

Mac OS X:

`None`

Windows:

`None`

Arguments

`file`	Is the name of the dynamic linker to be used.

Default

OFF	The default dynamic linker is used.

Description

This option lets you specify a dynamic linker other than the default.

Alternate Options

None

dynamiclib

Invokes the `libtool` *command to generate dynamic libraries.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux:

`None`

Mac OS X:

`-dynamiclib`

Windows:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option invokes the `libtool` command to generate dynamic libraries.

When passed this option, the compiler uses the `libtool` command to produce a dynamic library instead of an executable when linking.

To build static libraries, you should specify option `-staticlib` or `libtool -static <objects>`.

Alternate Options

None

See Also

-
- staticlib

dyncom, Qdyncom

Enables dynamic allocation of common blocks at run time.

IDE Equivalent

Windows: **Data > Dynamic Common Blocks**

Linux: None

Mac OS X: **Data > Dynamic Common Blocks**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-dyncom "`*`common1`*`,`*`common2`*`,..."`

Windows:

`/Qdyncom "`*`common1`*`,`*`common2`*`,..."`

Arguments

`common1``,`*`common2`*`,...`	Are the names of the common blocks to be dynamically allocated. The list of names must be within quotes.

Default

OFF	Common blocks are not dynamically allocated at run time.

Description

This option enables dynamic allocation of the specified common blocks at run time. For example, to enable dynamic allocation of common blocks a, b, and c at run time, use this syntax:

```
/Qdyncom "a,b,c"    ! on Windows systems

-dyncom "a,b,c"     ! on Linux and Mac OS X systems
```

The following are some limitations that you should be aware of when using this option:

- An entity in a dynamic common cannot be initialized in a DATA statement.
- Only named common blocks can be designated as dynamic COMMON.
- An entity in a dynamic common block must not be used in an EQUIVALENCE expression with an entity in a static common block or a DATA-initialized variable.

Alternate Options

None

See Also

-
-

Building Applications: Allocating Common Blocks

E

Causes the preprocessor to send output to `stdout`.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-E`

Windows:

`/E`

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`. Compilation stops when the files have been preprocessed.

When you specify this option, the compiler's preprocessor expands your source module and writes the result to `stdout`. The preprocessed source contains #line directives, which the compiler uses to determine the source file and line number.

Alternate Options

None

e90, e95, e03

See warn.

EP

Causes the preprocessor to send output to `stdout`*, omitting #line directives.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-EP`

Windows:

`/EP`

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the preprocessor to send output to `stdout`, omitting #line directives.

If you also specify option `preprocess-only`, option `P`, or option `F`, the preprocessor will write the results (without #line directives) to a file instead of `stdout`.

Alternate Options

None

error-limit

See diag-error-limit, Qdiag-error-limit.

exe

Specifies the name for a built program or dynamic-link library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/exe:{`*`file | dir`*`}`

Arguments

`file`	Is the name for the built program or dynamic-link library.
`dir`	Is the directory where the built program or dynamic-link library should be placed. It can include *`file`*.

Default

OFF	The name of the file is the name of the first source file on the command line with file extension .exe, so file.f becomes file.exe.

Description

This option specifies the name for a built program (.EXE) or a dynamic-link library (.DLL).

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

You can use this option to specify an alternate name for an executable file. This is especially useful when compiling and linking a set of input files. You can use the option to give the resulting file a name other than that of the first input file (source or object) on the command line.

Alternate Options

Linux and Mac OS X: `-o`

Windows: /Fe

Example

The following example creates a dynamic-link library file named file.dll (note that you can use `/LD` in place of `/dll`):

```
ifort /dll /exe:file.dll a.f
```

In the following example (which uses the alternate option `/Fe`), the command produces an executable file named outfile.exe as a result of compiling and linking three files: one object file and two Fortran source files.

```
prompt>ifort /Feoutfile.exe file1.obj file2.for file3.for
```

By default, this command produces an executable file named file1.exe.

See Also

-
- o

extend-source

Specifies the length of the statement field in a fixed-form source file.

IDE Equivalent

Windows: **Language > Fixed Form Line Length**

Linux: None

Mac OS X: **Language > Fixed Form Line Length**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-extend-source [size]`

`-noextend-source`

Windows:

`/extend-source[:size]`

`/noextend-source`

Arguments

size	Is the length of the statement field in a fixed-form source file. Possible values are: 72, 80, or 132.

Default

72	If you do not specify this option or you specify `noextend-source`, the statement field ends at column 72.
132	If you specify `extend_source` without *size*, the statement field ends at column 132.-

Description

This option specifies the size (column number) of the statement field of a source line in a fixed-form source file. This option is valid only for fixed-form files; it is ignored for free-form files.

When size is specified, it is the last column parsed as part of the statement field. Any columns after that are treated as comments.

If you do not specify *size*, it is the same as specifying `extend_source 132`.

Option	Description
`extend-source 72`	Specifies that the statement field ends at column 72.
`extend-source 80`	Specifies that the statement field ends at column 80.
`extend-source 132`	Specifies that the statement field ends at column 132.

Alternate Options

`extend-source 72`	Linux and Mac OS X: `-72` Windows: `/4L72`
`extend-source 80`	Linux and Mac OS X: `-80` Windows: `/4L80`
`extend-source 132`	Linux and Mac OS X: `-132` Windows: `/Qextend-source, /4L132`

extfor

Specifies file extensions to be processed by the compiler as Fortran files.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/extfor:`*`ext`*

Arguments

`ext`	Are the file extensions to be processed as a Fortran file.

Default

OFF	Only the file extensions recognized by the compiler are processed as Fortran files. For more information, see *Building Applications*.

Description

This option specifies file extensions (*`ext`*) to be processed by the compiler as Fortran files. It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extfor:myf95` and `/extfor:.myf95` are equivalent.

Alternate Options

None

See Also

-
- source compiler option

extfpp

Specifies file extensions to be recognized as a file to be preprocessed by the Fortran preprocessor.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/extfpp:`*`ext`*

Arguments

`ext`	Are the file extensions to be preprocessed by the Fortran preprocessor.

Default

OFF	Only the file extensions recognized by the compiler are preprocessed by `fpp`. For more information, see *Building Applications*.

Description

This option specifies file extensions (*`ext`*) to be recognized as a file to be preprocessed by the Fortran preprocessor (`fpp`). It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extfpp:myfpp` and `/extfpp:.myfpp` are equivalent.

Alternate Options

None

extlnk

Specifies file extensions to be passed directly to the linker.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/extlnk:`*`ext`*

Arguments

`ext`	Are the file extensions to be passed directly to the linker.

Default

OFF	Only the file extensions recognized by the compiler are passed to the linker. For more information, see Building Applications.

Description

This option specifies file extensions (*`ext`*) to be passed directly to the linker. It is useful if your source file has a nonstandard extension.

You can specify one or more file extensions. A leading period before each extension is optional; for example, `/extlnk:myobj` and `/extlnk:.myobj` are equivalent.

Alternate Options

None

F (Windows*)

Specifies the stack reserve amount for the program.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Fn`

Arguments

n	Is the stack reserve amount. It can be specified as a decimal integer or by using a C-style convention for constants (for example, /F0x1000).

Default

OFF	The stack size default is chosen by the operating system.

Description

This option specifies the stack reserve amount for the program. The amount (*n*) is passed to the linker.

Note that the linker property pages have their own option to do this.

Alternate Options

None

f66

Tells the compiler to apply FORTRAN 66 semantics.

IDE Equivalent

Windows: **Language > Enable FORTRAN 66 Semantics**

Linux: None

Mac OS X: **Language > Enable FORTRAN 66 Semantics**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-f66`

Windows:

`/f66`

Arguments

None

Default

OFF	The compiler applies Fortran 95 semantics.

Description

This option tells the compiler to apply FORTRAN 66 semantics when interpreting language features. This causes the following to occur:

- DO loops are always executed at least once
- FORTRAN 66 EXTERNAL statement syntax and semantics are allowed
- If the OPEN statement STATUS specifier is omitted, the default changes to STATUS='NEW' instead of STATUS='UNKNOWN'
- If the OPEN statement BLANK specifier is omitted, the default changes to BLANK='ZERO' instead of BLANK='NULL'

Alternate Options

Linux and Mac OS X: `-66`

Windows: None

f77rtl

Tells the compiler to use the run-time behavior of FORTRAN 77.

IDE Equivalent

Windows: **Compatibility > Enable F77 Run-Time Compatibility**

Linux: None

Mac OS X: **Compatibility > Enable F77 Run-Time Compatibility**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-f77rtl`

`-nof77rtl`

Windows:

`/f77rtl`

`/nof77rtl`

Arguments

None

Default

`nof77rtl`	The compiler uses the run-time behavior of Intel® Fortran.

Description

This option tells the compiler to use the run-time behavior of FORTRAN 77.

Specifying this option controls the following run-time behavior:

- When the unit is not connected to a file, some INQUIRE specifiers will return different values:
  - NUMBER= returns 0

  - ACCESS= returns 'UNKNOWN'
  - BLANK= returns 'UNKNOWN'
  - FORM= returns 'UNKNOWN'
- PAD= defaults to 'NO' for formatted input.
- NAMELIST and list-directed input of character strings must be delimited by apostrophes or quotes.
- When processing NAMELIST input:
  - Column 1 of each record is skipped.
  - The '$' or '&' that appears prior to the group-name must appear in column 2 of the input record.

Alternate Options

None

Fa

See asmfile

FA

See asmattr

falias

Determines whether aliasing should be assumed in the program.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-falias
```

`-fno-alias`

Windows:

`None`

Arguments

None

Default

`-falias`	Aliasing is assumed in the program.

Description

This option determines whether aliasing should be assumed in the program.

If you do not want aliasing to be assumed in the program, specify `-fno-alias`.

Alternate Options

Linux and Mac OS X: None

Windows: `/Oa[-]`

See Also

-
- `ffnalias`

falign-functions, Qfnalign

Tells the compiler to align functions on an optimal byte boundary.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-falign-functions[=n]`

`-fno-align-functions`

Windows:

`/Qfnalign[:n]`

`/Qfnalign-`

Arguments

n	Is the byte boundary for function alignment. Possible values are 2 or 16.

Default

`-fno-align-functions` or `/Qfnalign-`	The compiler aligns functions on 2-byte boundaries. This is the same as specifying `-falign-functions=2` (Linux and Mac OS X) or `/Qfnalign:2` (Windows).

Description

This option tells the compiler to align functions on an optimal byte boundary. If you do not specify *n*, the compiler aligns the start of functions on 16-byte boundaries.

Alternate Options

None

falign-stack

Tells the compiler the stack alignment to use on entry to routines.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux and Mac OS X:

`-falign-stack=`*mode*

Windows:

None

Arguments

mode	Is the method to use for stack alignment. Possible values are:
`default`	Tells the compiler to use default heuristics for stack alignment. If alignment is required, the compiler dynamically aligns the stack.
`maintain-16-byte`	Tells the compiler to not assume any specific stack alignment, but attempt to maintain alignment in case the stack is already aligned. If alignment is required, the compiler dynamically aligns the stack. This setting is compatible with GCC.
`assume-16-byte`	Tells the compiler to assume the stack is aligned on 16-byte boundaries and continue to maintain 16-byte alignment. This setting is compatible with GCC.

Default

`-falign-stack=default`	The compiler uses default heuristics for stack alignment.

Description

This option tells the compiler the stack alignment to use on entry to routines.

Alternate Options

None

fast

Maximizes speed across the entire program.

IDE Equivalent

Windows: None

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fast`

Windows:

`/fast`

Arguments

None

Default

OFF	The optimizations that maximize speed are not enabled.

Description

This option maximizes speed across the entire program.

It sets the following options:

- On systems using IA-64 architecture:
  Windows: `/O3` and `/Qipo`
  Linux: `-ipo`, `-O3`, and `-static`
- On systems using IA-32 architecture and Intel® 64 architecture:
  Mac OS X: `-ipo`, `-mdynamic-no-pic`, `-O3`, `-no-prec-div`, `-static`, and `-xHost`
  Windows: `/O3`, `/Qipo`, `/Qprec-div-`, and `/QxHost`
  Linux: `-ipo`, `-O3`, `-no-prec-div`, `-static`, and `-xHost`

When option `fast` is specified on systems using IA-32 architecture or Intel® 64 architecture, you can override the `-xHost` or `/QxHost` setting by specifying a different processor-specific `-x` or `/Qx` option on the command line. However, the last option specified on the command line takes precedence.

For example, if you specify `-fast -xSSE3` (Linux) or `/fast /QxSSE3` (Windows), option `-xSSE3` or `/QxSSE3` takes effect. However, if you specify `-xSSE3 -fast` (Linux) or `/QxSSE3 /fast` (Windows), option `-xHost` or `/QxHost` takes effect.

NOTE. The options set by option `fast` may change from release to release.

Alternate Options

None

See Also

-
- x, Qx

fast-transcendentals, Qfast-transcendentals

Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-fast-transcendentals
-no-fast-transcendentals
```

Windows:

`/Qfast-transcendentals`

`/Qfast-transcendentals-`

Default

`-fast-transcendentals` or `/Qfast-transcendentals`	The default depends on the setting of `-fp-model` (Linux and Mac OS X) or `/fp` (Windows). The default is ON if default setting `-fp-model fast` or `/fp:fast` is in effect. However, if a value-safe option such as `-fp-model precise` or `/fp:precise` is specified, the default is OFF.

Description

This option enables the compiler to replace calls to transcendental functions with implementations that may be faster but less precise.

It tells the compiler to perform certain optimizations on transcendental functions, such as replacing individual calls to sine in a loop with a single call to a less precise vectorized sine library routine.

This option has an effect only when specified with one of the following options:

- Windows* OS: `/fp:except` or `/fp:precise`
- Linux* OS and Mac OS* X: `-fp-model except` or `-fp-model precise`

You cannot use this option with option `-fp-model strict` (Linux and Mac OS X) or `/fp:strict` (Windows).

Alternate Options

None

See Also

-
-
- `fp-model, fp`

fcode-asm

Produces an assembly listing with machine code annotations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fcode-asm`

Windows:

`None`

Arguments

None

Default

OFF	No machine code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with machine code annotations.

The assembly listing file shows the hex machine instructions at the beginning of each line of assembly code. The file cannot be assembled; the filename is the name of the source file with an extension of `.cod`.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

Alternate Options

Linux and Mac OS X: None

Windows: `/asmattr:machine`

See Also

-
- S

Fe

See exe

fexceptions

Enables exception handling table generation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fexceptions`

`-fno-exceptions`

Windows:

`None`

Arguments

None

Default

-fno-exceptions	Exception handling table generation is disabled.

Description

This option enables C++ exception handling table generation, preventing Fortran routines in mixed-language applications from interfering with exception handling between C++ routines. The `-fno-exceptions` option disables C++ exception handling table generation, resulting in

smaller code. When this option is used, any use of C++ exception handling constructs (such as try blocks and throw statements) when a Fortran routine is in the call chain will produce an error.

Alternate Options

None

ffnalias

Specifies that aliasing should be assumed within functions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ffnalias`

`-fno-fnalias`

Windows:

`None`

Arguments

None

Default

-ffnalias	Aliasing is assumed within functions.

Description

This option specifies that aliasing should be assumed within functions.

The `-fno-fnalias` option specifies that aliasing should not be assumed within functions, but should be assumed across calls.

Alternate Options

Linux and Mac OS X: None

Windows: `/Ow[-]`

See Also

-
- falias

FI

See fixed.

finline

Tells the compiler to inline functions declared with cDEC$ ATTRIBUTES FORCEINLINE.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-finline`

`-fno-inline`

Windows:

`None`

Arguments

None

Default

`-fno-inline`	The compiler does not inline functions declared with cDEC$ ATTRIBUTES FORCEINLINE.

Description

This option tells the compiler to inline functions declared with cDEC$ ATTRIBUTES FORCEINLINE.

Alternate Options

None

finline-functions

Enables function inlining for single file compilation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-finline-functions`

`-fno-inline-functions`

Windows:

`None`

Arguments

None

Default

`-finline-functions`	Interprocedural optimizations occur. However, if you specify `-O0`, the default is OFF.

Description

This option enables function inlining for single file compilation.

It enables the compiler to perform inline function expansion for calls to functions defined within the current source file.

The compiler applies a heuristic to perform the function expansion. To specify the size of the function to be expanded, use the `-finline-limit` option.

Alternate Options

Linux and Mac OS X: `-inline-level=2`

Windows: `/Ob2`

See Also

-
- `ip`, `Qip`
- `finline-limit`
- Compiler Directed Inline Expansion of User Functions
- Inline Function Expansion

finline-limit

Lets you specify the maximum size of a function to be inlined.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-finline-limit=`*n*

Windows:

`None`

Arguments

n	Must be an integer greater than or equal to zero. It is the maximum number of lines the function can have to be considered for inlining.

Default

OFF	The compiler uses default heuristics when inlining functions.

Description

This option lets you specify the maximum size of a function to be inlined. The compiler inlines smaller functions, but this option lets you inline large functions. For example, to indicate a large function, you could specify 100 or 1000 for *n*.

Note that parts of functions cannot be inlined, only whole functions.

This option is a modification of the `-finline-functions` option, whose behavior occurs by default.

Alternate Options

None

See Also

-
- `finline-functions`

finstrument-functions, Qinstrument-functions

Determines whether routine entry and exit points are instrumented.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-finstrument-functions
-fno-instrument-functions
```

Windows:

```
/Qinstrument-functions
```

`/Qinstrument-functions-`

Arguments

None

Default

`-fno-instrument-functions` or `/Qinstrument-functions-`	Routine entry and exit points are not instrumented.

Description

This option determines whether routine entry and exit points are instrumented. It may increase execution time.

The following profiling functions are called with the address of the current routine and the address of where the routine was called (its "call site"):

- This function is called upon routine entry:
  - On IA-32 architecture and Intel® 64 architecture:

    ```
    void __cyg_profile_func_enter (void *this_fn,
    void *call_site);
    ```

  - On IA-64 architecture:

    ```
    void __cyg_profile_func_enter (void **this_fn,
    void *call_site);
    ```

- This function is called upon routine exit:
  - On IA-32 architecture and Intel® 64 architecture:

    ```
    void __cyg_profile_func_exit (void *this_fn,
    void *call_site);
    ```

  - On IA-64 architecture:

    ```
    void __cyg_profile_func_exit (void **this_fn,
    void *call_site);
    ```

On IA-64 architecture, the additional de-reference of the function pointer argument is required to obtain the routine entry point contained in the first word of the routine descriptor for indirect routine calls. The descriptor is documented in the Intel® Itanium® Software Conventions and Runtime Architecture Guide, section 8.4.2. You can find this design guide at web site http://www.intel.com.

These functions can be used to gather more information, such as profiling information or timing information. Note that it is the user's responsibility to provide these profiling functions.

If you specify `-finstrument-functions` (Linux and Mac OS X) or `/Qinstrument-functions` (Windows), routine inlining is disabled. If you specify `-fno-instrument-functions` or `/Qinstrument-functions-`, inlining is not disabled.

This option is provided for compatibility with gcc.

Alternate Options

None

fixed

Specifies source files are in fixed format.

IDE Equivalent

Windows: **Language > Source File Format** (`/free`, `/fixed`)

Linux: None

Mac OS X: **Language > Source File Format** (`/free`, `/fixed`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fixed`

`-nofixed`

Windows:

`/fixed`

`/nofixed`

Arguments

None

Default

OFF	The source file format is determined from the file extension.

Description

This option specifies source files are in fixed format. If this option is not specified, format is determined as follows:

- Files with an extension of .f90, .F90, or .i90 are free-format source files.
- Files with an extension of .f, .for, .FOR, .ftn, .FTN, .fpp, .FPP, or .i are fixed-format files.

Note that on Linux and Mac OS X systems, file names and file extensions are case sensitive.

Alternate Options

Linux and Mac OS X: `-FI`

Windows: `/nofree`, `/FI`, `/4Nf`

fkeep-static-consts, Qkeep-static-consts

Tells the compiler to preserve allocation of variables that are not referenced in the source.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-fkeep-static-consts
-fno-keep-static-consts
```

Windows:

`/Qkeep-static-consts`

`/Qkeep-static-consts-`

Arguments

None

Default

`-fno-keep-static-consts` or `/Qkeep-static-consts-`	If a variable is never referenced in a routine, the variable is discarded unless optimizations are disabled by option `-O0` (Linux and Mac OS X) or `/Od` (Windows).

Description

This option tells the compiler to preserve allocation of variables that are not referenced in the source.

The negated form can be useful when optimizations are enabled to reduce the memory usage of static data.

Alternate Options

None

fltconsistency

Enables improved floating-point consistency.

IDE Equivalent

Windows: **Floating-Point > Floating-Point Consistency**

Linux: None

Mac OS X: **Floating-Point > Improve Floating-Point Consistency**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fltconsistency`

`-nofltconsistency`

Windows:

`/fltconsistency`

`/nofltconsistency`

Arguments

None

Default

`nofltconsistency`	Improved floating-point consistency is not enabled. This setting provides better accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option enables improved floating-point consistency and may slightly reduce execution speed. It limits floating-point optimizations and maintains declared precision. It also disables inlining of math library functions.

Floating-point operations are not reordered and the result of each floating-point operation is stored in the target variable rather than being kept in the floating-point processor for use in a subsequent calculation.

For example, the compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. This change can alter the results of floating-point division computations slightly.

Floating-point intermediate results are kept in full 80 bits internal precision. Additionally, all spills/reloads of the X87 floating point registers are done using the internal formats; this prevents accidental loss of precision due to spill/reload behavior over which you have no control.

Specifying this option has the following effects on program compilation:

- On systems using IA-32 architecture or Intel® 64 architecture, floating-point user variables are not assigned to registers.

- On systems using IA-64 architecture, floating-point user variables may be assigned to registers. The expressions are evaluated using precision of source operands. The compiler will not use the Floating-point Multiply and Add (FMA) function to contract multiply and add/subtract operations in a single operation. The contractions can be enabled by using `-fma` (Linux) or `/Qfma` (Windows) option. The compiler will not speculate on floating-point operations that may affect the floating-point state of the machine.
- Floating-point arithmetic comparisons conform to IEEE 754.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.
- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform constant folding on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.
- Whenever an expression is spilled, it is spilled as 80 bits (extended precision), not 64 bits (DOUBLE PRECISION). When assignments to type REAL and DOUBLE PRECISION are made, the precision is rounded from 80 bits down to 32 bits (REAL) or 64 bits (DOUBLE PRECISION). When you do not specify `/Op`, the extra bits of precision are not always rounded away before the variable is reused.
- Even if vectorization is enabled by the `-x` (Linux and Mac OS X) or `/Qx` (Windows) options, the compiler does not vectorize reduction loops (loops computing the dot product) and loops with mixed precision types. Similarly, the compiler does not enable certain loop transformations. For example, the compiler does not transform reduction loops to perform partial summation or loop interchange.

This option causes performance degradation relative to using default floating-point optimization flags.

On Windows systems, an alternative is to use the `/Qprec` option, which should provide better than default floating-point precision while still delivering good floating-point performance.

The recommended method to control the semantics of floating-point calculations is to use option `-fp-model` (Linux and Mac OS X) or `/fp` (Windows).

Alternate Options

`fltconsistency`	Linux and Mac OS X: `-mp` (this is a deprecated option), `-mieee-fp`

	Windows: `/Op` (this is a deprecated option)
`nofltconsistency`	Linux and Mac OS X: `-mno-ieee-fp` Windows: None

See Also

-
- `mp1`, `Qprec`
- `fp-model`, `fp`

Building Applications: Using Compiler Optimizations

Fm

This option has been deprecated.
See map.

fma, Qfma

Enables the combining or contraction of floating-point multiplications and add or subtract operations.

IDE Equivalent

Windows: **Floating Point > Contract Floating-Point Operations**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

`-fma`

`-no-fma`

Mac OS X:

`None`

Windows:

`/Qfma`

`/Qfma-`

Arguments

None

Default

`-fma` or `/Qfma`	Floating-point multiplications and add/subtract operations are combined. However, if you specify `-fp-model strict` (Linux) or `/fp:strict` (Windows), but do not explicitly specify `-fma` or `/Qfma`, the default is `-no-fma` or `/Qfma-`.

Description

This option enables the combining or contraction of floating-point multiplications and add or subtract operations into a single operation.

Alternate Options

Linux: `-IPF-fma` (this is a deprecated option)

Windows: `/QIPF-fma` (this is a deprecated option)

See Also

-
-
- `fp-model, fp`

Floating-point Operations: Floating-point Options Quick Reference

fmath-errno

Tells the compiler that `errno` *can be reliably tested after calls to standard math library functions.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fmath-errno`

`-fno-math-errno`

Windows:

`None`

Arguments

None

Default

`-fno-math-errno`	The compiler assumes that the program does not test `errno` after calls to standard math library functions.

Description

This option tells the compiler to assume that the program tests `errno` after calls to math library functions. This restricts optimization because it causes the compiler to treat most math functions as having side effects.

Option `-fno-math-errno` tells the compiler to assume that the program does not test `errno` after calls to math library functions. This frequently allows the compiler to generate faster code. Floating-point code that relies on IEEE exceptions instead of `errno` to detect errors can safely use this option to improve performance.

Alternate Options

None

fminshared

Specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fminshared`

Windows:

`None`

Arguments

None

Default

OFF	Source files are compiled together to form a single object file.

Description

This option specifies that a compilation unit is a component of a main program and should not be linked as part of a shareable object.

This option allows the compiler to optimize references to defined symbols without special visibility settings. To ensure that external and common symbol references are optimized, you need to specify visibility hidden or protected by using the `-fvisibility`, `-fvisibility-hidden`, or `-fvisibility-protected` option.

Also, the compiler does not need to generate position-independent code for the main program. It can use absolute addressing, which may reduce the size of the global offset table (GOT) and may reduce memory traffic.

Alternate Options

None

See Also

-
- `fvisibility`

fnsplit, Qfnsplit

Enables function splitting.

IDE Equivalent

None

Architectures

`/Qfnsplit[-]`: IA-32 architecture, Intel® 64 architecture

`-[no-]fnsplit`: IA-64 architecture

Syntax

Linux:

```
-fnsplit
-no-fnsplit
```

Mac OS X:

```
None
```

Windows:

```
/Qfnsplit
/Qfnsplit-
```

Arguments

None

Default

`-no-fnsplit` or `/Qfnsplit-`	Function splitting is not enabled unless `-prof-use` (Linux) or `/Qprof-use` (Windows) is also specified.

Description

This option enables function splitting if `-prof-use` (Linux) or `/Qprof-use` (Windows) is also specified. Otherwise, this option has no effect.

It is enabled automatically if you specify `-prof-use` or `/Qprof-use`. If you do not specify one of those options, the default is `-no-fnsplit` (Linux) or `/Qfnsplit-` (Windows), which disables function splitting but leaves function grouping enabled.

To disable function splitting when you use `-prof-use` or `/Qprof-use`, specify `-no-fnsplit` or `/Qfnsplit-`.

Alternate Options

None

See Also

-
-
- Profile-Guided Optimization (PGO) Quick Reference
- Profile an Application

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

IDE Equivalent

Windows: **Optimization > Omit Frame Pointers**

Linux: None

Mac OS X: **Optimization > Provide Frame Pointer**

Architectures

`-f[no-]omit-frame-pointer`: IA-32 architecture, Intel® 64 architecture

`/Oy[-]`: IA-32 architecture

Syntax

Linux and Mac OS X:

`-fomit-frame-pointer`

`-fno-omit-frame-pointer`

Windows:

`/Oy`

`/Oy-`

Arguments

None

Default

`-fomit-frame-pointer` or `/Oy`	EBP is used as a general-purpose register in optimizations. However, on Linux* and Mac OS X systems, the default is `-fno-omit-frame-pointer` if option `-O0` or `-g` is specified. On Windows* systems, the default is `/Oy-` if option `/Od` is specified.

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Options `-fomit-frame-pointer` and `/Oy` allow this use. Options `-fno-omit-frame-pointer` and `/Oy-` disallow it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-fno-omit-frame-pointer` and `/Oy-` options direct the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For `-fno-omit-frame-pointer`: turning off optimizations with `-O0`
- For `/Oy-`: turning off `/O1`, `/O2`, or `/O3` optimizations

The `-fno-omit-frame-pointer` option is set when you specify option `-O0` or the `-g` option. The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3`.

The `/Oy` option is set when you specify the `/O1`, `/O2`, or `/O3` option. Option `/Oy-` is set when you specify the `/Od` option.

Using the `-fno-omit-frame-pointer` or `/Oy-` option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

Alternate Options

Linux and Mac OS X: `-fp` (this is a deprecated option)

Windows: None

Fo

See object

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

IDE Equivalent

Windows: **Optimization > Omit Frame Pointers**

Linux: None

Mac OS X: **Optimization > Provide Frame Pointer**

Architectures

`-f[no-]omit-frame-pointer`: IA-32 architecture, Intel® 64 architecture

`/Oy[-]`: IA-32 architecture

Syntax

Linux and Mac OS X:

```
-fomit-frame-pointer
-fno-omit-frame-pointer
```

Windows:

```
/Oy
/Oy-
```

Arguments

None

Default

`-fomit-frame-pointer` or `/Oy`	EBP is used as a general-purpose register in optimizations. However, on Linux* and Mac OS X systems, the default is `-fno-omit-frame-pointer` if option `-O0` or `-g` is specified. On Windows* systems, the default is `/Oy-` if option `/Od` is specified.

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Options `-fomit-frame-pointer` and `/Oy` allow this use. Options `-fno-omit-frame-pointer` and `/Oy-` disallow it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-fno-omit-frame-pointer` and `/Oy-` options direct the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For `-fno-omit-frame-pointer`: turning off optimizations with `-O0`
- For `/Oy-`: turning off `/O1`, `/O2`, or `/O3` optimizations

The `-fno-omit-frame-pointer` option is set when you specify option `-O0` or the `-g` option. The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3`.

The `/Oy` option is set when you specify the `/O1`, `/O2`, or `/O3` option. Option `/Oy-` is set when you specify the `/Od` option.

Using the `-fno-omit-frame-pointer` or `/Oy-` option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

Alternate Options

Linux and Mac OS X: `-fp` (this is a deprecated option)

Windows: None

fp-model, fp

Controls the semantics of floating-point calculations.

IDE Equivalent

Windows: Floating Point > Floating Point Model

Floating Point > Reliable Floating Point Exceptions Model

Linux: None

Mac OS X: **Floating Point > Floating Point Model**

Floating Point > Reliable Floating Point Exceptions Model

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fp-model` *keyword*

Windows:

`/fp:`*keyword*

Arguments

keyword Specifies the semantics to be used. Possible values are:

`precise`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
`strict`	Enables precise and except, disables contractions, and enables the property that allows modification of the floating-point environment.
`source`	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
`[no-]except` (Linux and Mac OS X) or `except[-]` (Windows)	Determines whether floating-point exception semantics are used.

Default

`-fp-model fast=1` or `/fp:fast=1`	The compiler uses more aggressive optimizations on floating-point calculations.

Description

This option controls the semantics of floating-point calculations.

The *keywords* can be considered in groups:

- Group A: `precise`, `fast`, `strict`
- Group B: `source`
- Group C: `except` (or the negative form)

You can use more than one *keyword*. However, the following rules apply:

- You cannot specify `fast` and `except` together in the same compilation. You can specify any other combination of group A, group B, and group C.
  Since `fast` is the default, you must not specify `except` without a group A or group B *keyword*.
- You should specify only one *keyword* from group A and only one keyword from group B. If you try to specify more than one *keyword* from either group A or group B, the last (rightmost) one takes effect.
- If you specify `except` more than once, the last (rightmost) one takes effect.

Option	Description
`-fp-model precise` or `/fp:precise`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance. The compiler assumes the default floating-point environment; you are not allowed to modify it.

Option	Description
	Floating-point exception semantics are disabled by default. To enable these semantics, you must also specify `-fp-model except` or `/fp:except`. For information on the semantics used to interpret floating-point calculations in the source code, see `precise` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.
`-fp-model fast[=1|2]` or `/fp:fast[=1|2]`	Tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed, but may alter the accuracy of floating-point computations. Specifying fast is the same as specifying `fast=1`. `fast=2` may produce faster and less accurate results. Floating-point exception semantics are disabled by default and they cannot be enabled because you cannot specify fast and except together in the same compilation. To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details). For information on the semantics used to interpret floating-point calculations in the source code, see `fast` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.
`-fp-model strict` or `/fp:strict`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

Option	Description
	The compiler does not assume the default floating-point environment; you are allowed to modify it. Floating-point exception semantics can be disabled by explicitly specifying `-fp-model no-except` or `/fp:except-`. For information on the semantics used to interpret floating-point calculations in the source code, see `strict` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.
`-fp-model source` or `/fp:source`	This option causes intermediate results to be rounded to the precision defined in the source code. It also implies keyword `precise` unless it is overridden by a keyword from Group A. The compiler assumes the default floating-point environment; you are not allowed to modify it. For information on the semantics used to interpret floating-point calculations in the source code, see `source` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.

NOTE. This option cannot be used to change the default (source) precision for the calculation of intermediate results.

NOTE. On Windows and Linux operating systems on IA-32 architecture, the compiler, by default, implements floating-point (FP) arithmetic using SSE2 and SSE instructions. This can cause differences in floating-point results when compared to previous x87 implementations.

Alternate Options

None

Example

For examples of how to use this option, see *Floating-point Operations: Using the* `-fp-model (/fp)` *Option*.

See Also

-
-
- `O`
- `Od`
- `mp1, Qprec`
- Floating-point Environment

The MSDN article Microsoft Visual C++ Floating-Point Optimization, which discusses concepts that apply to this option.

fp-model, fp

Controls the semantics of floating-point calculations.

IDE Equivalent

Windows: Floating Point > Floating Point Model

Floating Point > Reliable Floating Point Exceptions Model

Linux: None

Mac OS X: **Floating Point > Floating Point Model**

Floating Point > Reliable Floating Point Exceptions Model

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fp-model` *keyword*

Windows:

`/fp:keyword`

Arguments

keyword	Specifies the semantics to be used. Possible values are:	
	`precise`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision.
	`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
	`strict`	Enables precise and except, disables contractions, and enables the property that allows modification of the floating-point environment.
	`source`	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
	`[no-]except` (Linux and Mac OS X) or `except[-]` (Windows)	Determines whether floating-point exception semantics are used.

Default

`-fp-model fast=1` or `/fp:fast=1`	The compiler uses more aggressive optimizations on floating-point calculations.

Description

This option controls the semantics of floating-point calculations.

The *keywords* can be considered in groups:

- Group A: `precise`, `fast`, `strict`
- Group B: `source`
- Group C: `except` (or the negative form)

You can use more than one *keyword*. However, the following rules apply:

- You cannot specify `fast` and `except` together in the same compilation. You can specify any other combination of group A, group B, and group C.
  Since `fast` is the default, you must not specify `except` without a group A or group B *keyword*.
- You should specify only one *keyword* from group A and only one keyword from group B. If you try to specify more than one *keyword* from either group A or group B, the last (rightmost) one takes effect.
- If you specify `except` more than once, the last (rightmost) one takes effect.

Option	Description
`-fp-model precise` or `/fp:precise`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance. The compiler assumes the default floating-point environment; you are not allowed to modify it. Floating-point exception semantics are disabled by default. To enable these semantics, you must also specify `-fp-model except` or `/fp:except`. For information on the semantics used to interpret floating-point calculations in the source code, see `precise` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.
`-fp-model fast[=1|2]` or `/fp:fast[=1|2]`	Tells the compiler to use more aggressive optimizations when implementing floating-point calculations. These optimizations increase speed, but may alter the accuracy of floating-point computations.

Option	Description
	Specifying fast is the same as specifying `fast=1`. `fast=2` may produce faster and less accurate results. Floating-point exception semantics are disabled by default and they cannot be enabled because you cannot specify fast and except together in the same compilation. To enable exception semantics, you must explicitly specify another keyword (see other keyword descriptions for details). For information on the semantics used to interpret floating-point calculations in the source code, see `fast` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.
`-fp-model strict` or `/fp:strict`	Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model. The compiler does not assume the default floating-point environment; you are allowed to modify it. Floating-point exception semantics can be disabled by explicitly specifying `-fp-model no-except` or `/fp:except-`. For information on the semantics used to interpret floating-point calculations in the source code, see `strict` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.

Option	Description
`-fp-model source` or `/fp:source`	This option causes intermediate results to be rounded to the precision defined in the source code. It also implies keyword `precise` unless it is overridden by a keyword from Group A. The compiler assumes the default floating-point environment; you are not allowed to modify it. For information on the semantics used to interpret floating-point calculations in the source code, see `source` in *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.

NOTE. This option cannot be used to change the default (source) precision for the calculation of intermediate results.

NOTE. On Windows and Linux operating systems on IA-32 architecture, the compiler, by default, implements floating-point (FP) arithmetic using SSE2 and SSE instructions. This can cause differences in floating-point results when compared to previous x87 implementations.

Alternate Options

None

Example

For examples of how to use this option, see *Floating-point Operations: Using the* `-fp-model` *(*`/fp`*) Option*.

See Also

-
-
- ○

- Od
- mp1, Qprec
- Floating-point Environment

The MSDN article Microsoft Visual C++ Floating-Point Optimization, which discusses concepts that apply to this option.

fp-port, Qfp-port

Rounds floating-point results after floating-point operations.

IDE Equivalent

Windows: **Floating-Point > Round Floating-Point Results**

Linux: None

Mac OS X: **Floating-Point > Round Floating-Point Results**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fp-port`

`-no-fp-port`

Windows:

`/Qfp-port`

`/Qfp-port-`

Arguments

None

Default

`-no-fp-port` or `/Qfp-port-`	The default rounding behavior depends on the compiler's code generation decisions and the precision parameters of the operating system.

Description

This option rounds floating-point results after floating-point operations. Rounding to user-specified precision occurs at assignments and type conversions. This has some impact on speed.

The default is to keep results of floating-point operations in higher precision. This provides better performance but less consistent floating-point results.

Alternate Options

None

fp-relaxed, Qfp-relaxed

Enables use of faster but slightly less accurate code sequences for math functions.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-fp-relaxed
-no-fp-relaxed
```

Mac OS X:

```
None
```

Windows:

```
/Qfp-relaxed
/Qfp-relaxed-
```

Arguments

None

Default

`-no-fp-relaxed` or `/Qfp-relaxed-`	Default code sequences are used for math functions.

Description

This option enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt. When compared to strict IEEE* precision, this option slightly reduces the accuracy of floating-point calculations performed by these functions, usually limited to the least significant digit.

This option also enables the performance of more aggressive floating-point transformations, which may affect accuracy.

Alternate Options

Linux: `-IPF-fp-relaxed` (this is a deprecated option)

Windows: `/QIPF-fp-relaxed` (this is a deprecated option)

See Also

-
-
- `fp-model, fp`

fp-speculation, Qfp-speculation

Tells the compiler the mode in which to speculate on floating-point operations.

IDE Equivalent

Windows: **Floating Point > Floating-Point Speculation**

Linux: None

Mac OS X: **Floating Point > Floating-Point Speculation**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fp-speculation=`*mode*

Windows:

`/Qfp-speculation:`*mode*

Arguments

mode	Is the mode for floating-point operations. Possible values are:
`fast`	Tells the compiler to speculate on floating-point operations.
`safe`	Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception.
`strict`	Tells the compiler to disable speculation on floating-point operations.
`off`	This is the same as specifying strict.

Default

`-fp-speculation=fast` or `/Qfp-speculation:fast`	The compiler speculates on floating-point operations. This is also the behavior when optimizations are enabled. However, if you specify no optimizations (`-O0` on Linux; `/Od` on Windows), the default is `-fp-speculation=safe` (Linux) or `/Qfp-speculation:safe` (Windows).

Description

This option tells the compiler the mode in which to speculate on floating-point operations.

Alternate Options

None

See Also

-
-

fp-stack-check, Qfp-stack-check

Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.

IDE Equivalent

Windows: **Floating-Point > Check Floating-point Stack**

Linux: None

Mac OS X: **Floating-Point > Check Floating-point Stack**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-fp-stack-check
```

Windows:

```
/Qfp-stack-check
```

Arguments

None

Default

OFF	There is no checking to ensure that the floating-point (FP) stack is in the expected state.

Description

This option tells the compiler to generate extra code after every function call to ensure that the floating-point (FP) stack is in the expected state.

By default, there is no checking. So when the FP stack overflows, a NaN value is put into FP calculations and the program's results differ. Unfortunately, the overflow point can be far away from the point of the actual bug. This option places code that causes an access violation exception immediately after an incorrect call occurs, thus making it easier to locate these issues.

Alternate Options

None

See Also

•
•

Floating-point Operations:

- Checking the Floating-point Stack State

fpconstant

Tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.

IDE Equivalent

Windows: **Floating-Point > Extend Precision of Single-Precision Constants**

Linux: None

Mac OS X: **Floating-Point > Extend Precision of Single-Precision Constants**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fpconstant`

`-nofpconstant`

Windows:

`/fpconstant`

`/nofpconstant`

Arguments

None

Default

`nofpconstant`	Single-precision constants assigned to double-precision variables are evaluated in single precision according to Fortran 95/90 Standard rules.

Description

This option tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.

This is extended precision. It does not comply with the Fortran 95/90 standard, which requires that single-precision constants assigned to double-precision variables be evaluated in single precision.

It allows compatibility with FORTRAN 77, where such extended precision was allowed. If this option is not used, certain programs originally created for FORTRAN 77 compilers may show different floating-point results because they rely on the extended precision for single-precision constants assigned to double-precision variables.

Alternate Options

None

Example

In the following example, if you specify `fpconstant`, identical values are assigned to D1 and D2. If you omit `fpconstant`, the compiler will obey the Fortran 95/90 Standard and assign a less precise value to D1:

```
REAL (KIND=8) D1, D2
DATA D1 /2.71828182846182/ ! REAL (KIND=4) value expanded to double
DATA D2 /2.71828182846182D0/ ! Double value assigned to double
```

fpe

Allows some control over floating-point exception handling for the main program at run-time.

IDE Equivalent

Windows: **Floating-Point > Floating-Point Exception Handling**

Linux: None

Mac OS X: **Floating-Point > Floating-Point Exception Handling**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fpe`*n*

Windows:

`/fpe:`*n*

Arguments

n	Specifies the floating-point exception handling level. Possible values are:
`0`	Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is aborted. This option sets the `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows) option; therefore underflow results will be set to zero unless you explicitly specify `-no-ftz` (Linux and Mac OS X) or `/Qftz-` (Windows). On systems using IA-64 architecture, underflow behavior is equivalent to specifying option `-ftz` or `/Qftz`. On systems using IA-32 architecture or Intel® 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero. By contrast, option `-ftz` or `/Qftz` only sets SSE underflow results to zero. To get more detailed location information about where the error occurred, use option `traceback`.

`1`	All floating-point exceptions are disabled. On systems using IA-64 architecture, underflow behavior is equivalent to specifying option `-ftz` or `/Qftz`. On systems using IA-32 architecture or Intel® 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero.
`3`	All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero, such as `-ftz` or `/Qftz`, `O3`, or `O2` on systems using IA-32 architecture or Intel® 64 architecture. This setting provides full IEEE support.

Default

`-fpe3` or `/fpe:3`	All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero.

Description

This option allows some control over floating-point exception handling for the main program at run-time. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported.

The `fpe` option affects how the following conditions are handled:

- When floating-point calculations result in a divide by zero, overflow, or invalid operation.
- When floating-point calculations result in an underflow.
- When a denormalized number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression.

When enabled exceptions occur, execution is aborted and the cause of the abort reported to the user. If compiler option `traceback` is specified at compile time, detailed information about the location of the abort is also reported.

This option does not enable underflow exceptions, input denormal exceptions, or inexact exceptions.

Alternate Options

None

See Also

-
- `fpe-all`
- `ftz`, `Qftz`
- `traceback`
- Using the -fpe or /fpe Compiler Option

fpe-all

Allows some control over floating-point exception handling for each routine in a program at run-time.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fpe-all=`*n*

Windows:

`/fpe-all:`*n*

Arguments

n	Specifies the floating-point exception handling level. Possible values are:
`0`	Floating-point invalid, divide-by-zero, and overflow exceptions are enabled. If any such exceptions occur, execution is

	aborted. This option sets the `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows) option; therefore underflow results will be set to zero unless you explicitly specify `-no-ftz` (Linux and Mac OS X) or `/Qftz-` (Windows). On systems using IA-64 architecture, underflow behavior is equivalent to specifying option `-ftz` or `/Qftz`. On systems using IA-32 architecture or Intel® 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero. By contrast, option `-ftz` or `/Qftz` only sets SSE underflow results to zero. To get more detailed location information about where the error occurred, use option `traceback`.
`1`	All floating-point exceptions are disabled. On systems using IA-64 architecture, underflow behavior is equivalent to specifying option `-ftz` or `/Qftz`. On systems using IA-32 architecture or Intel® 64 architecture, underflow results from SSE instructions, as well as x87 instructions, will be set to zero.
`3`	All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero, such as `-ftz` or `/Qftz`, `O3`, or `O2` on systems using IA-32 architecture or Intel® 64 architecture. This setting provides full IEEE support.

Default

`-fpe-all=3` or `/fpe-all:3` or the setting of `fpe` that the main program was compiled with	All floating-point exceptions are disabled. Floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero.

Description

This option allows some control over floating-point exception handling for each routine in a program at run-time. This includes whether exceptional floating-point values are allowed and how precisely run-time exceptions are reported.

The `fpe-all` option affects how the following conditions are handled:

- When floating-point calculations result in a divide by zero, overflow, or invalid operation.
- When floating-point calculations result in an underflow.
- When a denormalized number or other exceptional number (positive infinity, negative infinity, or a NaN) is present in an arithmetic expression.

The current settings of the floating-point exception and status flags are saved on each routine entry and restored on each routine exit. This may incur some performance overhead.

When option `fpe-all` is applied to a main program, it has the same affect as when option `fpe` is applied to the main program.

When enabled exceptions occur, execution is aborted and the cause of the abort reported to the user. If compiler option `traceback` is specified at compile time, detailed information about the location of the abort is also reported.

This option does not enable underflow exceptions, input denormal exceptions, or inexact exceptions.

Option `fpe-all` sets option `assume ieee_fpe_flags`.

Alternate Options

None

See Also

-
- `assume`
- `fpe`

- ftz, Qftz
- traceback
- Using the -fpe or /fpe Compiler Option

fpic

Determines whether the compiler generates position-independent code.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fpic`

`-fno-pic`

Windows:

`None`

Arguments

None

Default

`-fno-pic` or `-fpic`	On systems using IA-32 or Intel® 64 architecture, the compiler does not generate position-independent code. On systems using IA-64 architecture, the compiler generates position-independent code.

Description

This option determines whether the compiler generates position-independent code.

Option `-fpic` specifies full symbol preemption. Global symbol definitions as well as global symbol references get default (that is, preemptable) visibility unless explicitly specified otherwise.

Option `-fno-pic` is only valid on systems using IA-32 or Intel® 64 architecture.

On systems using IA-32 or Intel® 64 architecture, `-fpic` must be used when building shared objects.

This option can also be specified as `-fPIC`.

Alternate Options

None

fpie

Tells the compiler to generate position-independent code. The generated code can only be linked into executables.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-fpie`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	The compiler does not generate position-independent code for an executable-only object.

Description

This option tells the compiler to generate position-independent code. It is similar to `-fpic`, but code generated by `-fpie` can only be linked into an executable.

Because the object is linked into an executable, this option causes better optimization of some symbol references.

To ensure that run-time libraries are set up properly for the executable, you should also specify option `-pie` to the compiler driver on the link command line.

Option `-fpie` can also be specified as `-fPIE`.

Alternate Options

None

See Also

-
- fpic
- pie

fpp, Qfpp

Runs the Fortran preprocessor on source files before compilation.

IDE Equivalent

Windows: **Preprocessor > Preprocess Source File**

Linux: None

Mac OS X: Preprocessor > Preprocess Source File

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`fpp[`*`n`*`]`

`fpp[="`*`option`*`"]`

`-nofpp`

Windows:

`/fpp[n]`

`/fpp[:"option"]`

`/nofpp`

`/Qfpp[n]`

`/Qfpp[:"option"]`

Arguments

n	Deprecated. Tells the compiler whether to run the preprocessor or not. Possible values are:
	`0` Tells the compiler not to run the preprocessor.
	`1, 2, or 3` Tells the compiler to run the preprocessor.
option	Is a Fortran preprocessor (fpp) option; for example, "-macro=no", which disables macro expansion. The quotes are required. For a list of fpp options, see *Fortran Preprocessor Options*.

Default

`nofpp`	The Fortran preprocessor is not run on files before compilation.

Description

This option runs the Fortran preprocessor on source files before they are compiled.

If the option is specified with no argument, the compiler runs the preprocessor.

If `0` is specified for *n*, it is equivalent to `nofpp`.Note that argument *n* is deprecated.

We recommend you use option `Qoption,fpp,"option"` to pass fpp options to the Fortran preprocessor.

Alternate Options

Linux and Mac OS X: `-cpp`

Windows: `/Qcpp`

See Also

-
-
- Fortran Preprocessor Options
- Qoption

fpscomp

Controls whether certain aspects of the run-time system and semantic language features within the compiler are compatible with Intel® Fortran or Microsoft Fortran PowerStation.*

IDE Equivalent

Windows: **Compatibility > Use Filenames from Command Line** (`/fpscomp:[no]filesfromcmd`)

Compatibility > Use PowerStation I/O Format (`/fpscomp:[no]ioformat`)

Compatibility > Use PowerStation Portability Library (`/fpscomp:[no]libs`)

Compatibility > Use PowerStation List-Directed I/O Spacing (`/fpscomp:[no]ldio_spacing`)

Compatibility > Use PowerStation Logical Values (`/fpscomp:[no]logicals`)

Compatibility > Use Other PowerStation Run-Time Behavior (`/fpscomp:[no]general`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fpscomp [keyword]`

`-nofpscomp`

Windows:

`/fpscomp[:keyword]`

/nofpscomp

Arguments

keyword Specifies the compatibility that the compiler should follow. Possible values are:

none	Specifies that no options should be used for compatibility.
[no]filesfromcmd	Determines what compatibility is used when the OPEN statement FILE= specifier is blank.
[no]general	Determines what compatibility is used when semantics differences exist between Fortran PowerStation and Intel® Fortran.
[no]ioformat	Determines what compatibility is used for list-directed formatted and unformatted I/O.
[no]libs	Determines whether the portability library is passed to the linker.
[no]ldio_spacing	Determines whether a blank is inserted at run-time after a numeric value before a character value.
[no]logicals	Determines what compatibility is used for representation of LOGICAL values.
all	Specifies that all options should be used for compatibility.

Default

fpscomp libs The portability library is passed to the linker.

Description

This option controls whether certain aspects of the run-time system and semantic language features within the compiler are compatible with Intel Fortran or Microsoft* Fortran PowerStation.

If you experience problems when porting applications from Fortran PowerStation, specify fpscomp (or fpscomp all). When porting applications from Intel Fortran, use fpscomp none or fpscomp libs (the default).

Option	Description
`fpscomp none`	Specifies that no options should be used for compatibility with Fortran PowerStation. This is the same as specifying `nofpscomp`.Option `fpscomp none` enables full Intel® Fortran compatibility. If you omit `fpscomp`, the default is `fpscomp libs`. You cannot use the `fpscomp` and `vms` options in the same command.
`fpscomp filesfromcmd`	Specifies Fortran PowerStation behavior when the OPEN statement FILE= specifier is blank (FILE=' '). It causes the following actions to be taken at run-time: • The program reads a filename from the list of arguments (if any) in the command line that invoked the program. If any of the command-line arguments contain a null string (''), the program asks the user for the corresponding filename. Each additional OPEN statement with a blank FILE= specifier reads the next command-line argument. • If there are more nameless OPEN statements than command-line arguments, the program prompts for additional file names. • In a QuickWin application, a **File Select** dialog box appears to request file names. To prevent the run-time system from using the filename specified on the command line when the OPEN statement FILE specifier is omitted, specify `fpscomp nofilesfromcmd`. This allows the application of Intel Fortran defaults, such as the FORTn environment variable and the FORT.*n* file name (where *n* is the unit number). The `fpscomp filesfromcmd` option affects the following Fortran features: • The OPEN statement FILE specifier For example, assume a program OPENTEST contains the following statements: OPEN(UNIT = 2, FILE = ' ') OPEN(UNIT = 3, FILE = ' ') OPEN(UNIT = 4, FILE = ' ')

Option	Description
	The following command line assigns the file TEST.DAT to unit 2, prompts the user for a filename to associate with unit 3, then prompts again for a filename to associate with unit 4: opentest test.dat '' '' • Implicit file open statements such as the WRITE, READ, and ENDFILE statements Unopened files referred to in READ or WRITE statements are opened implicitly as if there had been an OPEN statement with a name specified as all blanks. The name is read from the command line.

`fpscomp general`	Specifies that Fortran PowerStation semantics should be used when a difference exists between Intel Fortran and Fortran PowerStation. The `fpscomp general` option affects the following Fortran features:
	
	• The BACKSPACE statement:
	
	  • It allows files opened with ACCESS='APPEND' to be used with the BACKSPACE statement.
	
	  • It allows files opened with ACCESS='DIRECT' to be used with the BACKSPACE statement.
	
	  Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be used with the BACKSPACE statement violates the Fortran 95 standard and may be removed in the future.
	
	• The READ statement:
	
	• • It causes a READ from a formatted file opened for direct access to read records that have the same record type format as Fortran PowerStation. This consists of accounting for the trailing Carriage Return/Line Feed pair (<CR><LF>) that is part of the record. It allows sequential reads from a formatted file opened for direct access.
	
	  Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be used with the sequential READ statement violates the Fortran 95 standard and may be removed in the future.

Option	Description
	• It allows the last record in a file opened with FORM='FORMATTED' and a record type of STREAM_LF or STREAM_CR that does not end with a proper record terminator (<line feed> or <carriage return>) to be read without producing an error. • It allows sequential reads from an unformatted file opened for direct access. • Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be read with the sequential READ statement violates the Fortran 95 standard and may be removed in the future. • The INQUIRE statement: • • The CARRIAGECONTROL specifier returns the value "UNDEFINED" instead of "UNKNOWN" when the carriage control is not known. • The NAME specifier returns the file name "UNKNOWN" instead of filling the file name with spaces when the file name is not known. • The SEQUENTIAL specifier returns the value "YES" instead of "NO" for a direct access formatted file. • The UNFORMATTED specifier returns the value "NO" instead of "UNKNOWN" when it is not known whether unformatted I/O can be performed to the file. Note: Returning the value "NO" instead of "UNKNOWN" for this specifier violates the Fortran 95 standard and may be removed in the future. • The OPEN statement: • • If a file is opened with an unspecified STATUS keyword value, and is not named (no FILE specifier), the file is opened as a scratch file. For example: OPEN (UNIT = 4) • In contrast, when fpscomp nogeneral is in effect with an unspecified STATUS value with no FILE specifier, the FORTn environment variable and the FORT.n file name are used (where n is the unit number).

Option	Description
	• If the STATUS value was not specified and if the name of the file is "USER", the file is marked for deletion when it is closed. • It allows a file to be opened with the APPEND and READONLY characteristics. • If the default for the CARRIAGECONTROL specifier is assumed, it gives "LIST" carriage control to direct access formatted files instead of "NONE". • If the default for the CARRIAGECONTROL specifier is assumed and the device type is a terminal file, the file is given the default carriage control value of "FORTRAN" instead of "LIST". • It gives an opened file the additional default of write sharing. • It gives the file a default block size of 1024 instead of 8192. • If the default for the MODE and ACTION specifier is assumed and there was an error opening the file, try opening the file as read only, then write only. • If a file that is being re-opened has a different file type than the current existing file, an error is returned. • It gives direct access formatted files the same record type as Fortran PowerStation. This means accounting for the trailing Carriage Return/Line Feed pair (<CR><LF>) that is part of the record. • The STOP statement: It writes the Fortran PowerStation output string and/or returns the same exit condition values. • The WRITE statement: • • Writing to formatted direct files When writing to a formatted file opened for direct access, records are written in the same record type format as Fortran PowerStation. This consists of adding the trailing Carriage Return/Line Feed pair <CR><LF>) that is part of the record. It ignores the CARRIAGECONTROL specifier setting when writing to a formatted direct access file. • Interpreting Fortran carriage control characters

Option	Description
	When interpreting Fortran carriage control characters during formatted I/O, carriage control sequences are written that are the same as Fortran PowerStation. This is true for the "Space, 0, 1 and + " characters. • Performing non-advancing I/O to the terminal When performing non-advancing I/O to the terminal, output is written in the same format as Fortran PowerStation. • Interpreting the backslash (\) and dollar ($) edit descriptors When interpreting backslash and dollar edit descriptors during formatted I/O, sequences are written the same as Fortran PowerStation. • Performing sequential writes It allows sequential writes from an unformatted file opened for direct access. Note: Allowing files that are not opened with sequential access (such as ACCESS='DIRECT') to be read with the sequential WRITE statement violates the Fortran 95 standard and may be removed in the future. Specifying `fpscomp general` sets `fpscomp ldio_spacing`.
`fpscomp ioformat`	Specifies that Fortran PowerStation semantic conventions and record formats should be used for list-directed formatted and unformatted I/O. The fpscomp ioformat option affects the following Fortran features: • The WRITE statement: • • For formatted list-directed WRITE statements, formatted internal list-directed WRITE statements, and formatted namelist WRITE statements, the output line, field width values, and the list-directed data type semantics are determined according to the following sample for real constants (N below): For 1 <= N < 10**7, use F15.6 for single precision or F24.15 for double.

Option	Description

For N < 1 or N >= 10**7, use E15.6E2 for single precision or E24.15E3 for double.

See the Fortran PowerStation documentation for more detailed information about the other data types affected.

- For unformatted WRITE statements, the unformatted file semantics are dictated according to the Fortran PowerStation documentation; these semantics are different from the Intel Fortran file format. See the Fortran PowerStation documentation for more detailed information.

The following table summarizes the default output formats for list-directed output with the intrinsic data types:

Data Type	Output Format with fpscomp noioformat	Output Format with fpscomp ioformat
BYTE	I5	I12
LOGICAL (all)	L2	L2
INTEGER(1)	I5	I12
INTEGER(2)	I7	I12
INTEGER(4)	I12	I12
INTEGER(8)	I22	I22
REAL(4)	1PG15.7E2	1PG16.6E2
REAL(8)	1PG24.15E3	1PG25.15E3
COMPLEX(4)	'(',1PG14.7E2, ', ',1PG14.7E2, ') '	'(',1PG16.6E2, ', ',1PG16.6E2, ') '
COMPLEX(8)	'(',1PG23.15E3, ', ',1PG23.15E3, ') '	'(',1PG25.15E3, ', ',1PG25.15E3, ') '

Option	Description

Data Type	Output Format with fpscomp noioformat	Output Format with fpscomp ioformat
CHARACTER	Aw	Aw

- The READ statement:
- For formatted list-directed READ statements, formatted internal list-directed READ statements, and formatted namelist READ statements, the field width values and the list-directed semantics are dictated according to the following sample for real constants (N below):

  For 1 <= N < 10**7, use F15.6 for single precision or F24.15 for double.

  For N < 1 or N >= 10**7, use E15.6E2 for single precision or E24.15E3 for double.

  See the Fortran PowerStation documentation for more detailed information about the other data types affected.
- For unformatted READ statements, the unformatted file semantics are dictated according to the Fortran PowerStation documentation; these semantics are different from the Intel Fortran file format. See the Fortran PowerStation documentation for more detailed information.

`fpscomp nolibs` — Prevents the portability library from being passed to the linker.

`fpscomp ldio_spacing` — Specifies that at run time a blank should not be inserted after a numeric value before a character value (undelimited character string). This representation is used by Intel Fortran releases before Version 8.0 and by Fortran PowerStation. If you specify `fpscomp general`, it sets `fpscomp ldio_spacing`.

`fpscomp logicals` — Specifies that integers with a non-zero value are treated as true, integers with a zero value are treated as false. The literal constant .TRUE. has an integer value of 1, and the literal constant .FALSE. has an integer value of 0. This representation is used by Intel Fortran releases before Version

Option	Description
	8.0 and by Fortran PowerStation. The default is `fpscomp nologicals`, which specifies that odd integer values (low bit one) are treated as true and even integer values (low bit zero) are treated as false. The literal constant .TRUE. has an integer value of -1, and the literal constant .FALSE. has an integer value of 0. This representation is used by Compaq* Visual Fortran. The internal representation of LOGICAL values is not specified by the Fortran standard. Programs which use integer values in LOGICAL contexts, or which pass LOGICAL values to procedures written in other languages, are non-portable and may not execute correctly. Intel recommends that you avoid coding practices that depend on the internal representation of LOGICAL values. The `fpscomp logical` option affects the results of all logical expressions and affects the return value for the following Fortran features: • The INQUIRE statement specifiers OPENED, IOFOCUS, EXISTS, and NAMED • The EOF intrinsic function • The BTEST intrinsic function • The lexical intrinsic functions LLT, LLE, LGT, and LGE
`fpscomp all`	Specifies that all options should be used for compatibility with Fortran PowerStation. This is the same as specifying `fpscomp` with no keyword. Option `fpscomp all` enables full compatibility with Fortran PowerStation.

Alternate Options

None

See Also

•

Building Applications: Microsoft Fortran PowerStation Compatible Files

FR

See free.

fr32

Disables the use of the high floating-point registers.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-fr32`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	The use of the high floating-point registers is enabled.

Description

This option disables the use of the high floating-point registers. Only the lower 32 floating-point registers are used.

Alternate Options

None

free

Specifies source files are in free format.

IDE Equivalent

Windows: **Language > Source File Format** (`/free`, `/fixed`)

Linux: None

Mac OS X: **Language > Source File Format** (`/free`, `/fixed`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-free`

`-nofree`

Windows:

`/free`

`/nofree`

Arguments

None

Default

OFF	The source file format is determined from the file extension.

Description

This option specifies source files are in free format. If this option is not specified, format is determined as follows:

- Files with an extension of .f90, .F90, or .i90 are free-format source files.
- Files with an extension of .f, .for, .FOR, .ftn, or .i are fixed-format files.

Alternate Options

Linux and Mac OS X: `-FR`

Windows: `/nofixed`, `/FR`, `/4Yf`

See Also

-
- `fixed`

fsource-asm

Produces an assembly listing with source code annotations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fsource-asm`

Windows:

`None`

Arguments

None

Default

OFF	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with source code annotations. The assembly listing file shows the source code as interspersed comments.

To use this option, you must also specify option `-S`, which causes an assembly listing to be generated.

Alternate Options

Linux and Mac OS X: None

Windows: `/asmattr:source`, `/FAs`

See Also

-
- `S`

fstack-security-check, GS

Determines whether the compiler generates code that detects some buffer overruns.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-fstack-security-check`

`-fno-stack-security-check`

Windows:

`/GS`

`/GS-`

Arguments

None

Default

`-fno-stack-security-check`	The compiler does not detect buffer overruns.

or /GS-

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

The /GS option is supported with Microsoft Visual Studio .NET 2003* and Microsoft Visual Studio 2005*.

Alternate Options

Linux and Mac OS X: -f[no-]stack-protector

Windows: None

fstack-security-check, GS

Determines whether the compiler generates code that detects some buffer overruns.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

-fstack-security-check

-fno-stack-security-check

Windows:

/GS

/GS-

Arguments

None

Default

`-fno-stack-security-check` or `/GS-`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

The `/GS` option is supported with Microsoft Visual Studio .NET 2003* and Microsoft Visual Studio 2005*.

Alternate Options

Linux and Mac OS X: `-f[no-]stack-protector`

Windows: None

fsyntax-only

See syntax-only.

ftrapuv, Qtrapuv

Initializes stack local variables to an unusual value to aid error detection.

IDE Equivalent

Windows: **Data > Initialize stack variables to an unusual value**

Linux: None

Mac OS X: **Run-Time > Initialize Stack Variables to an Unusual Value**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ftrapuv`

Windows:

`/Qtrapuv`

Arguments

None

Default

OFF	The compiler does not initialize local variables.

Description

This option initializes stack local variables to an unusual value to aid error detection. Normally, these local variables should be initialized in the application.

The option sets any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors.

This option sets option `-g` (Linux and Mac OS X) and `/Zi` or `/Z7` (Windows).

Alternate Options

None

See Also

-
-
- `g`, `Zi`, `Z7`

ftz, Qftz

Flushes denormal results to zero.

IDE Equivalent

Windows: (IA-32 and IA-64 architectures): **Floating Point > Flush Denormal Results to Zero**

(Intel® 64 architecture): None

Linux: None

Mac OS X: **Floating Point > Flush Denormal Results to Zero**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ftz`

`-no-ftz`

Windows:

`/Qftz`

`/Qftz-`

Arguments

None

Default

Systems using IA-64 architecture: `-no-ftz` or `/Qftz-` Systems using IA-32 architecture and Intel® 64 architecture: `-ftz` or `/Qftz`	On systems using IA-64 architecture, the compiler lets results gradually underflow. On systems using IA-32 architecture and Intel® 64 architecture, denormal results are flushed to zero.

Description

This option flushes denormal results to zero when the application is in the gradual underflow mode. It may improve performance if the denormal values are not critical to your application's behavior.

This option sets or resets the FTZ and the DAZ hardware flags. If FTZ is ON, denormal results from floating-point calculations will be set to the value zero. If FTZ is OFF, denormal results remain as is. If DAZ is ON, denormal values used as input to floating-point instructions will be treated as zero. If DAZ is OFF, denormal instruction inputs remain as is. Systems using IA-64 architecture have FTZ but not DAZ. Systems using Intel® 64 architecture have both FTZ and DAZ. FTZ and DAZ are not supported on all IA-32 architectures.

When `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows) is used in combination with an SSE-enabling option on systems using IA-32 architecture (for example, `xN` or `QxN`), the compiler will insert code in the main routine to set FTZ and DAZ. When `-ftz` or `/Qftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check. `-no-ftz` (Linux and Mac OS X) or `/Qftz-` (Windows) will prevent the compiler from inserting any code that might set FTZ or DAZ.

This option only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in FTZ/DAZ mode.

Options `-fpe0` and `-fpe1` (Linux and Mac OS X) set `-ftz`. Options `/fpe:0` and `/fpe:1` (Windows) set `/Qftz`.

On systems using IA-64 architecture, optimization option `O3` sets `-ftz` and `/Qftz`; optimization option `O2` sets `-no-ftz` (Linux) and `/Qftz-` (Windows). On systems using IA-32 architecture and Intel® 64 architecture, every optimization option `O` level, except `O0`, sets `-ftz` and `/Qftz`.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by using `-no-ftz` or `/Qftz-` in the command line while still benefiting from the `O3` optimizations.

NOTE. Options `-ftz` and `/Qftz` are performance options. Setting these options does not guarantee that all denormals in a program are flushed to zero. They only cause denormals generated at run time to be flushed to zero.

Alternate Options

None

See Also

-
-
- `x, Qx`

Floating-point Operations: Using the -fpe or /fpe Compiler Option

func-groups

This is a deprecated option. See prof-func-groups.

funroll-loops

See unroll, Qunroll.

fverbose-asm

Produces an assembly listing with compiler comments, including options and version information.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fverbose-asm`

`-fno-verbose-asm`

Windows:

`None`

Arguments

None

Default

`-fno-verbose-asm`	No source code annotations appear in the assembly listing file, if one is produced.

Description

This option produces an assembly listing file with compiler comments, including options and version information.

To use this option, you must also specify `-S`, which sets `-fverbose-asm`.

If you do not want this default when you specify `-S`, specify `-fno-verbose-asm`.

Alternate Options

None

See Also

-
- `S`

fvisibility

Specifies the default visibility for global symbols or the visibility for symbols in a file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fvisibility=`*`keyword`*

`-fvisibility-`*`keyword=file`*

Windows:

`None`

Arguments

`keyword`	Specifies the visibility setting. Possible values are:	
	`default`	Sets visibility to default.
	`extern`	Sets visibility to extern.
	`hidden`	Sets visibility to hidden.
	`internal`	Sets visibility to internal.

`protected`	Sets visibility to protected.
file	Is the pathname of a file containing the list of symbols whose visibility you want to set. The symbols must be separated by whitespace (spaces, tabs, or newlines).

Default

-fvisibility=default	The compiler sets visibility of symbols to default.

Description

This option specifies the default visibility for global symbols (syntax `-fvisibility=`*keyword*) or the visibility for symbols in a file (syntax `-fvisibility-`*keyword*`=`*file*).

Visibility specified by `-fvisibility-`*keyword*`=`*file* overrides visibility specified by `-fvisibility=`*keyword* for symbols specified in a file.

Option	Description
`-fvisibility=default` `-fvisibility-default=`*file*	Sets visibility of symbols to default. This means other components can reference the symbol, and the symbol definition can be overridden (preempted) by a definition of the same name in another component.
`-fvisibility=extern` `-fvisibility-extern=`*file*	Sets visibility of symbols to extern. This means the symbol is treated as though it is defined in another component. It also means that the symbol can be overridden by a definition of the same name in another component.
`-fvisibility=hidden` `-fvisibility-hidden=`*file*	Sets visibility of symbols to hidden. This means that other components cannot directly reference the symbol. However, its address may be passed to other components indirectly.

Option	Description
`-fvisibility=internal` `-fvisibility-internal=`*`file`*	Sets visibility of symbols to internal. This means the symbol cannot be referenced outside its defining component, either directly or indirectly.
`-fvisibility=protected` `-fvisibility-protected=`*`file`*	CELL_TEXT

If an `-fvisibility` option is specified more than once on the command line, the last specification takes precedence over any others.

If a symbol appears in more than one visibility *`file`*, the setting with the least visibility takes precedence.

The following shows the precedence of the visibility settings (from greatest to least visibility):

- `extern`
- `default`
- `protected`
- `hidden`
- `internal`

Note that `extern` visibility only applies to functions. If a variable symbol is specified as `extern`, it is assumed to be `default`.

Alternate Options

None

Example

A file named `prot.txt` contains symbols a, b, c, d, and e. Consider the following:

```
-fvisibility-protected=prot.txt
```

This option sets `protected` visibility for all the symbols in the file. It has the same effect as specifying `fvisibility=protected` in the declaration for each of the symbols.

See Also

-

- Symbol Visibility Attribute Options (Linux* and Mac OS* X)

g, Zi, Z7

Tells the compiler to generate full debugging information in the object file.

IDE Equivalent

Windows: **General > Debug Information Format** (`/Z7, /Zd, /Zi`)

Linux: None

Mac OS X: **General > Generate Debug Information** (`-g`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-g`

Windows:

`/Zi`

`/Z7`

Arguments

None

Default

OFF	No debugging information is produced in the object file.

Description

This option tells the compiler to generate symbolic debugging information in the object file for use by debuggers.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off `O2` and makes `O0` (Linux and Mac OS X) or `Od` (Windows) the default unless `O2` (or another `O` option) is explicitly specified in the same command line.

On Linux systems using Intel® 64 architecture and Linux and Mac OS X systems using IA-32 architecture, specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option.

For more information on `Zi` and `Z7`, see keyword `full` in debug (Windows*).

Alternate Options

Linux and Mac OS X: None

Windows: `/debug:full` (or `/debug`)

See Also

-
-
-
- `Zd`

G2, G2-p9000

Optimizes application performance for systems using IA-64 architecture.

IDE Equivalent

Windows: **Optimization > Optimize For Intel® Processor**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/G2`

`/G2-p9000`

Arguments

None

Default

`/G2-p9000`	Performance is optimized for Dual-Core Intel® Itanium® 2 processor 9000 series.

Description

These options optimize application performance for a particular Intel® processor or family of processors. The compiler generates code that takes advantage of features of IA-64 architecture.

Option	Description
`G2`	Optimizes for Intel® Itanium® 2 processors.
`G2-p9000`	Optimizes for Dual-Core Intel® Itanium® 2 processor 9000 series. This option affects the order of the generated instructions, but the generated instructions are limited to Intel® Itanium® 2 processor instructions unless the program specifies and executes intrinsics specific to the Dual-Core Intel® Itanium® 2 processor 9000 series.

The resulting executable is backwards compatible and generated code is optimized for specific processors. For example, code generated with `/G2-p9000` will run correctly on single-core Itanium® 2 processors, but it might not run as fast as if it had been generated using `/G2`.

Alternate Options

`/G2`	Linux: `-mtune=itanium2` Mac OS X: None Windows: None

`/G2-p9000`	Linux: `-mtune=itanium2-p9000`, `-mcpu=itanium2-p9000` (-mcpu is a deprecated option) Mac OS X: None Windows: None

Example

In the following example, the compiled binary of the source program prog.f is optimized for the Dual-Core Intel® Itanium® 2 processor 9000 series by default. The same binary will also run on single-core Itanium® 2 processors (unless the program specifies and executes intrinsics specific to the Dual-Core Intel® Itanium® 2 processor 9000 series). All lines in the code example are equivalent.

```
ifort prog.f
```

```
ifort /G2-p9000 prog.f
```

In the following example, the compiled binary is optimized for single-core Itanium® 2 processors:

```
ifort /G2 prog.f
```

See Also

-
- `mtune`

G5, G6, G7

Optimize application performance for systems using IA-32 architecture and Intel® 64 architecture. These are deprecated options.

IDE Equivalent

Windows: **Optimization > Optimize For Intel(R) Processor** (`/GB`, `/G5`, `/G6`, `/G7`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/G5`

`/G6`

`/G7`

Arguments

None

Default

/G7	On systems using IA-32 architecture and Intel® 64 architecture, performance is optimized for Intel® Pentium® 4 processors, Intel® Xeon® processors, Intel® Pentium® M processors, and Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support.

Description

These options optimize application performance for a particular Intel® processor or family of processors. The compiler generates code that takes advantage of features of the specified processor.

Option	Description
G5	Optimizes for Intel® Pentium® and Pentium® with MMX™ technology processors.
G6	Optimizes for Intel® Pentium® Pro, Pentium® II and Pentium® III processors.
G7	Optimizes for Intel® Core™ Duo processors, Intel® Core™ Solo processors, Intel® Pentium® 4 processors, Intel® Xeon® processors based on the Intel® Core microarchitecture, Intel® Pentium® M processors, and Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3) instruction support.

On systems using Intel® 64 architecture, only option G7 is valid.

These options always generate code that is backwards compatible with Intel processors of the same architecture. For example, code generated with the G7 option runs correctly on Pentium III processors, although performance may be faster on Pentium III processors when compiled using or G6.

Alternate Options

Windows: `/GB` (an alternate for `/G6`; this option is also deprecated)

Linux: None

Example

In the following example, the compiled binary of the source program prog.f is optimized, by default, for Intel® Pentium® 4 processors, Intel® Xeon® processors, Intel® Pentium® M processors, and Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3). The same binary will also run on Pentium, Pentium Pro, Pentium II, and Pentium III processors. All lines in the code example are equivalent.

```
ifort prog.f
```

```
ifort /G7 prog.f
```

In the following example, the compiled binary is optimized for Pentium processors and Pentium processors with MMX technology:

```
ifort /G5 prog.f
```

```
icl /G5 prog.c
```

See Also

-
- `mtune`

gdwarf-2

Enables generation of debug information using the DWARF2 format.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-gdwarf-2`

Windows:

`None`

Arguments

None

Default

OFF	No debug information is generated. However, if compiler option `-g` is specified, debug information is generated in the latest DWARF format, which is currently DWARF2.

Description

This option enables generation of debug information using the DWARF2 format. This is currently the default when compiler option `-g` is specified.

Alternate Options

None

See Also

-
- `g`

Ge

Enables stack-checking for all functions.
This option has been deprecated.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Ge`

Arguments

None

Default

OFF	Stack-checking for all functions is disabled.

Description

This option enables stack-checking for all functions.

Alternate Options

None

gen-interfaces

Tells the compiler to generate an interface block for each routine in a source file.

IDE Equivalent

Windows: **Diagnostics > Generate Interface Blocks**

Linux: None

Mac OS X: **Diagnostics > Generate Interface Blocks**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-gen-interfaces [[no]source]`

`-nogen-interfaces`

Windows:

`/gen-interfaces[:[no]source]`

`/nogen-interfaces`

Arguments

None

Default

`nogen-interfaces`	The compiler does not generate interface blocks for routines in a source file.

Description

This option tells the compiler to generate an interface block for each routine (that is, for each SUBROUTINE and FUNCTION statement) defined in the source file. The compiler generates two files for each routine, a .mod file and a .f90 file, and places them in the current directory or in the directory specified by the include (`-I`) or `-module` option. The .f90 file is the text of the interface block; the .mod file is the interface block compiled into binary form.

If `source` is specified, the compiler creates the *_mod.f90 as well as the *_mod.mod files. If `nosource` is specified, the compiler creates the *_mod.mod but not the *_mod.f90 files. If neither is specified, it is the same as specifying `-gen-interfaces source` (Linux and Mac OS X) or `/gen-interfaces:source` (Windows).

Alternate Options

None

global-hoist, Qglobal-hoist

Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-global-hoist`

`-no-global-hoist`

Windows:

`/Qglobal-hoist`

`/Qglobal-hoist-`

Arguments

None

Default

`-global-hoist` or `/Qglobal-hoist`	Certain optimizations are enabled that can move memory loads.

Description

This option enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source. In most cases, these optimizations are safe and can improve performance.

The `-no-global-hoist` (Linux and Mac OS X) or `/Qglobal-hoist-` (Windows) option is useful for some applications, such as those that use shared or dynamically mapped memory, which can fail if a load is moved too early in the execution stream (for example, before the memory is mapped).

Alternate Options

None

Gm

See keyword `cvf` *in iface.*

Gs

Disables stack-checking for routines with more than a specified number of bytes of local variables and compiler temporaries.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Gs[n]`

Arguments

n	Is the number of bytes of local variables and compiler temporaries.

Default

`4096`	Stack checking is disabled for routines with more than 4KB of stack space allocated.

Description

This option disables stack-checking for routines with *n* or more bytes of local variables and compiler temporaries. If you do not specify *n*, you get the default of 4096.

Alternate Options

None

fstack-security-check, GS

Determines whether the compiler generates code that detects some buffer overruns.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-fstack-security-check`

`-fno-stack-security-check`

Windows:

`/GS`

`/GS-`

Arguments

None

Default

`-fno-stack-security-check` or `/GS-`	The compiler does not detect buffer overruns.

Description

This option determines whether the compiler generates code that detects some buffer overruns that overwrite the return address. This is a common technique for exploiting code that does not enforce buffer size restrictions.

The `/GS` option is supported with Microsoft Visual Studio .NET 2003* and Microsoft Visual Studio 2005*.

Alternate Options

Linux and Mac OS X: `-f[no-]stack-protector`

Windows: None

Gz

See keyword `stdcall` *in iface*

heap-arrays

Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.

IDE Equivalent

Windows: **Optimization > Heap Arrays**

Linux: None

Mac OS X: **Optimization > Heap Arrays**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-heap-arrays [size]`

`-no-heap-arrays`

Windows:

`/heap-arrays[:size]`

`/heap-arrays-`

Arguments

size	Is an integer value representing the size of the arrays in kilobytes. Any arrays known at compile-time to be larger than *size* are allocated on the heap instead of the stack.

Default

`-no-heap-arrays` or `/heap-arrays-`	The compiler puts automatic arrays and arrays created for temporary computations in temporary storage in the stack storage area.

Description

This option puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.

If `heap-arrays` is specified and *size* is omitted, all automatic and temporary arrays are put on the heap. If 10 is specified for *size*, all automatic and temporary arrays larger than 10 KB are put on the heap.

Alternate Options

None

Example

In Fortran, an automatic array gets it size from a run-time expression. For example:

```
RECURSIVE SUBROUTINE F( N )
INTEGER :: N
REAL :: X ( N )     ! an automatic array
REAL :: Y ( 1000 )  ! an explicit-shape local array on the stack
```

Array X in the example above is affected by the `heap-array` option. Array Y is not.

help

Displays all available compiler options or a category of compiler options.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-help[`*category*`]`

Windows:

`/help[`*category*`]`

Arguments

category Is a category or class of options to display. Possible values are:

`advanced`	Displays advanced optimization options that allow fine tuning of compilation or allow control over advanced features of the compiler.
`codegen`	Displays Code Generation options.
`compatibility`	Displays options affecting language compatibility.
`component`	Displays options for component control.
`data`	Displays options related to interpretation of data in programs or the storage of data.
`deprecated`	Displays options that have been deprecated.
`diagnostics`	Displays options that affect diagnostic messages displayed by the compiler.
`float`	Displays options that affect floating-point operations.
`help`	Displays all the available help categories.
`inline`	Displays options that affect inlining.
`ipo`	Displays Interprocedural Optimization (IPO) options

`language`	Displays options affecting the behavior of the compiler language features.
`link`	Displays linking or linker options.
`misc`	Displays miscellaneous options that do not fit within other categories.
`openmp`	Displays OpenMP and parallel processing options.
`opt`	Displays options that help you optimize code.
`output`	Displays options that provide control over compiler output.
`pgo`	Displays Profile Guided Optimization (PGO) options.
`preproc`	Displays options that affect preprocessing operations.
`reports`	Displays options for optimization reports.

Default

OFF	No list is displayed unless this compiler option is specified.

Description

This option displays all available compiler options or a category of compiler options. If category is not specified, all available compiler options are displayed.

Alternate Options

Linux and Mac OS X: None

Windows: `/?`

homeparams

Tells the compiler to store parameters passed in registers to the stack.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/homeparams`

Arguments

None

Default

OFF	Register parameters are not written to the stack.

Description

This option tells the compiler to store parameters passed in registers to the stack.

Alternate Options

None

hotpatch

Tells the compiler to prepare a routine for hotpatching.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/hotpatch[:n]`

Arguments

n	An integer specifying the number of bytes the compiler should add before the function entry point.

Default

OFF	The compiler does not prepare routines for hotpatching.

Description

This option tells the compiler to prepare a routine for hotpatching. The compiler inserts nop padding around function entry points so that the resulting image is hot patchable.

Specifically, the compiler adds nop bytes after each function entry point and enough nop bytes before the function entry point to fit a direct jump instruction on the target architecture.

If *n* is specified, it overrides the default number of bytes that the compiler adds before the function entry point.

Alternate Options

None

I

Specifies an additional directory for the include path.

IDE Equivalent

Windows: **General > Additional Include Directories** (/include)

Preprocessor > Additional Include Directories (/include)

Linux: None

Mac OS X: Preprocessor > Additional Include Directories (/include)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-I`*dir*

Windows:

`/I`*dir*

Arguments

dir	Is the directory to add to the include path.

Default

OFF	The default include path is used.

Description

This option specifies an additional directory for the include path, which is searched for module files referenced in USE statements and include files referenced in INCLUDE statements. To specify multiple directories on the command line, repeat the option for each directory you want to add.

For all USE statements and for those INCLUDE statements whose file name does not begin with a device or directory name, the directories are searched in this order:

1. The directory containing the first source file.

   Note that if `assume nosource_include` is specified, this directory will not be searched.
2. The current working directory where the compilation is taking place (if different from the above directory).
3. Any directory or directories specified using the I option. If multiple directories are specified, they are searched in the order specified on the command line, from left to right.
4. On Linux and Mac OS X systems, any directories indicated using environment variable FPATH. On Windows systems, any directories indicated using environment variable INCLUDE.

This option affects fpp preprocessor behavior and the USE statement.

Alternate Options

Linux and Mac OS X: None

Windows: `/include`

See Also

-
- `X`
- `assume`

i-dynamic

This is a deprecated option. See shared-intel.

i-static

This is a deprecated option. See static-intel.

i2, i4, i8

See integer-size.

idirafter

Adds a directory to the second include file search path.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-idirafter`*`dir`*

Windows:

`None`

Arguments

`dir`	Is the name of the directory to add.

Default

OFF	Include file search paths include certain default directories.

Description

This option adds a directory to the second include file search path (after `-I`).

Alternate Options

None

iface

Specifies the default calling convention and argument-passing convention for an application.

IDE Equivalent

Windows: **External Procedures > Calling Convention** (`/iface:{cref|stdref|std-call|cvf|default}`)

External Procedures > String Length Argument Passing (`/iface:[no]mixed_str_len_arg`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/iface:`*`keyword`*

Arguments

keyword	Specifies the calling convention or the argument-passing convention. Possible values are:

`default`	Tells the compiler to use the default calling conventions.
`cref`	Tells the compiler to use calling conventions C, REFERENCE.
`cvf`	Tells the compiler to use calling conventions compatible with Compaq Visual Fortran*.
`[no]mixed_str_len_arg`	Determines the argument-passing convention for hidden-length character arguments.
`stdcall`	Tells the compiler to use calling convention STDCALL.
`stdref`	Tells the compiler to use calling conventions STDCALL, REFERENCE.

Default

`/iface:default`	The default calling convention is used.

Description

This option specifies the default calling convention and argument-passing convention for an application.

The aspects of calling and argument passing controlled by this option are as follows:

- The calling mechanism (C or STDCALL): On IA-32 architecture, these mechanisms differ in how the stack register is adjusted when a procedure call returns. On Intel® 64 and IA-64 architectures, the only calling mechanism available is C; requests for the STDCALL mechanism are ignored.
- The argument passing mechanism (by value or by reference)
- Character-length argument passing (at the end of the argument list or after the argument address)
- The case of external names (uppercase or lowercase)
- The name decoration (prefix and suffix)

You can also use the ATTRIBUTES compiler directive to modify these conventions on an individual basis. Note that the effects of the ATTRIBUTES directive do not always match that of the iface option of the same name.

Option	Description
/iface:default	Tells the compiler to use the default calling conventions. These conventions are as follows: • The calling mechanism: C • The argument passing mechanism: by reference • Character-length argument passing: at end of argument list • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 or IA-64 architecture; no suffix
/iface:cref	Tells the compiler to use the same conventions as /iface:default except that external names are lowercase.
/iface:cvf	Tells the compiler to use calling conventions compatible with Compaq Visual Fortran* and Microsoft Fortran PowerStation. These conventions are as follows: • The calling mechanism: STDCALL • The argument passing mechanism: by reference • Character-length argument passing: following the argument address • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 or IA-64 architecture. On Windows* systems using IA-32 architecture, @n suffix where *n* is the number of bytes to be removed from the stack on exit from the procedure. No suffix on other systems.
/iface:mixed_str_len_arg	Specifies argument-passing conventions for hidden-length character arguments. This option tells the compiler that the hidden length passed for a character argument is to be placed immediately after its corresponding character argument in the argument list.

Option	Description
	This is the method used by Compaq Visual Fortran*. When porting mixed-language programs that pass character arguments, either this option must be specified correctly or the order of hidden length arguments must be changed in the source code. This option can be used in addition to other `/iface` options.
`/iface:std-call`	Tells the compiler to use the following conventions: • The calling mechanism: STDCALL • The argument passing mechanism: by value • Character-length argument passing: at the end of the argument list • The external name case: uppercase • The name decoration: Underscore prefix on IA-32 architecture, no prefix on Intel® 64 or IA-64 architecture. On Windows* systems using IA-32 architecture, @n suffix where *n* is the number of bytes to be removed from the stack on exit from the procedure. No suffix on other systems.
`/iface:stdref`	Tells the compiler to use the same conventions as `/iface:stdcall` except that argument passing is by reference.

CAUTION. On Windows systems, if you specify option `/iface:cref`, it overrides the default for external names and causes them to be lowercase. It is as if you specified "!dec$ attributes c, reference" for the external name.

If you specify option `/iface:cref` and want external names to be uppercase, you must explicitly specify option `/names:uppercase`.

CAUTION. On systems using IA-32 architecture, there must be agreement between the calling program and the called procedure as to which calling mechanism (C or STDCALL) is used or unpredictable errors may occur. If you change the default mechanism to STDCALL, you must use the ATTRIBUTES DEFAULT directive to reset the calling conventions for routines specified with the USEROPEN keyword in an OPEN statement and for comparison routines passed to the QSORT library routine.

Alternate Options

`/iface:cvf`	Linux and Mac OS X: None Windows: `/Gm`
`/iface:mixed_str_len_arg`	Linux and Mac OS X: `-mixed-str-len-arg` Windows: None
`/iface:nomixed_str_len_arg`	Linux and Mac OS X: `-nomixed-str-len-arg` Windows: None
`/iface:stdcall`	Linux and Mac OS X: None Windows: `/Gz`

See Also

•

Building Applications: Programming with Mixed Languages Overview

Language Reference: ATTRIBUTES

implicitnone

See warn.

include

See I.

inline

Specifies the level of inline function expansion.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/inline[:keyword]`

Arguments

keyword	Is the level of inline function expansion. Possible values are:	
	`none`	Disables inlining of user-defined functions. This is the same as specifying `manual`.
	`manual`	Disables inlining of user-defined functions. Fortran statement functions are always inlined.
	size	Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and most speed optimizations.
	`speed`	Enables inlining of any function. This is the same as specifying `all`.
	`all`	Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and all speed optimizations. This is the same as specifying `inline` with no *keyword*.

Default

OFF	The compiler inlines certain functions by default.

Description

This option specifies the level of inline function expansion.

Alternate Options

`inline all` or `inline speed`	Linux and Mac OS X: None Windows: `/Ob2 /Ot`

`inline size`	Linux and Mac OS X: None Windows: `/Ob2 /Os`
`inline manual`	Linux and Mac OS X: None Windows: `/Ob0`
`inline none`	Linux and Mac OS X: None Windows: `/Ob0`

See Also

-
- `finline-functions`

inline-debug-info, Qinline-debug-info

Produces enhanced source position information for inlined code. This is a deprecated option.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-inline-debug-info
```

Windows:

```
/Qinline-debug-info
```

Arguments

None

Default

OFF	No enhanced source position information is produced for inlined code.

Description

This option produces enhanced source position information for inlined code. This leads to greater accuracy when reporting the source location of any instruction. It also provides enhanced debug information useful for function call traceback.

To use this option for debugging, you must also specify a debug enabling option, such as `-g` (Linux) or `/debug` (Windows).

Alternate Options

Linux and Mac OS X: `-debug inline-debug-info`

Windows: None

inline-factor, Qinline-factor

Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-factor=`*n*

`-no-inline-factor`

Windows:

`/Qinline-factor=`*n*

`/Qinline-factor-`

Arguments

n	Is a positive integer specifying the percentage value. The default value is 100 (a factor of 1).

Default

`-no-inline-factor` or `/Qinline-factor-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the percentage multiplier that should be applied to all inlining options that define upper limits:

- `-inline-max-size` and `/Qinline-max-size`
- `-inline-max-total-size` and `/Qinline-max-total-size`
- `-inline-max-per-routine` and `/Qinline-max-per-routine`
- `-inline-max-per-compile` and `/Qinline-max-per-compile`

This option takes the default value for each of the above options and multiplies it by *n* divided by 100. For example, if 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2. This option is useful if you do not want to individually increase each option limit.

If you specify `-no-inline-factor` (Linux and Mac OS X) or `/Qinline-factor-` (Windows), the following occurs:

- Every function is considered to be a small or medium function; there are no large functions.
- There is no limit to the size a routine may grow when inline expansion is performed.
- There is no limit to the number of times some routine may be inlined into a particular routine.
- There is no limit to the number of times inlining can be applied to a compilation unit.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase default limits, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-max-size`, `Qinline-max-size`
- `inline-max-total-size`, `Qinline-max-total-size`
- `inline-max-per-routine`, `Qinline-max-per-routine`
- `inline-max-per-compile`, `Qinline-max-per-compile`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-forceinline, Qinline-forceinline

Specifies that an inline routine should be inlined whenever the compiler can do so.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-inline-forceinline
```

Windows:

```
/Qinline-forceinline
```

Default

OFF	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies that a inline routine should be inlined whenever the compiler can do so. This causes the routines marked with an inline keyword or directive to be treated as if they were "forceinline".

NOTE. Because C++ member functions whose definitions are included in the class declaration are considered inline functions by default, using this option will also make these member functions "forceinline" functions.

The "forceinline" condition can also be specified by using the directive cDEC$ ATTRIBUTES FORCEINLINE.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS) or `/Qopt-report` (Windows).

CAUTION. When you use this option to change the meaning of inline to "forceinline", the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-level, Ob

Specifies the level of inline function expansion.

IDE Equivalent

Windows: **Optimization > Inline Function Expansion**

Linux: None

Mac OS X: **Optimization > Inline Function Expansion**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-level=`*n*

Windows:

`/Ob`*n*

Arguments

n	Is the inline function expansion level. Possible values are 0, 1, and 2.

Default

`-inline-level=2` or `/Ob2`	This is the default if option `O2` is specified or is in effect by default. On Windows systems, this is also the default if option `O3` is specified.
`-inline-level=0` or `/Ob0`	This is the default if option `-O0` (Linux and Mac OS) or `/Od` (Windows) is specified.

Description

This option specifies the level of inline function expansion. Inlining procedures can greatly improve the run-time performance of certain programs.

Option	Description
`-inline-level=0` or `Ob0`	Disables inlining of user-defined functions. Note that statement functions are always inlined.
`-inline-level=1` or `Ob1`	Enables inlining when an inline keyword or an inline directive is specified.
`-inline-level=2` or `Ob2`	Enables inlining of any function at the compiler's discretion.

Alternate Options

Linux: `-Ob` (this is a deprecated option)

Mac OS X: None

Windows: None

See Also

-
-
- `inline`

inline-max-per-compile, Qinline-max-per-compile

Specifies the maximum number of times inlining may be applied to an entire compilation unit.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-per-compile=`*n*

`-no-inline-max-per-compile`

Windows:

`/Qinline-max-per-compile=`*n*

`/Qinline-max-per-compile-`

Arguments

n	Is a positive integer that specifies the number of times inlining may be applied.

Default

`-no-inline-max-per-compile` or `/Qinline-max-per-compile-`	The compiler uses default heuristics for inline routine expansion.

Description

This option the maximum number of times inlining may be applied to an entire compilation unit. It limits the number of times that inlining can be applied.

For compilations using Interprocedural Optimizations (IPO), the entire compilation is a compilation unit. For other compilations, a compilation unit is a file.

If you specify `-no-inline-max-per-compile` (Linux and Mac OS X) or `/Qinline-max-per-compile-` (Windows), there is no limit to the number of times inlining may be applied to a compilation unit.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-per-routine, Qinline-max-per-routine

Specifies the maximum number of times the inliner may inline into a particular routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-per-routine=`*n*

`-no-inline-max-per-routine`

Windows:

`/Qinline-max-per-routine=`*n*

`/Qinline-max-per-routine-`

Arguments

n	Is a positive integer that specifies the maximum number of times the inliner may inline into a particular routine.

Default

`-no-inline-max-per-routine` or `/Qinline-max-per-routine-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the maximum number of times the inliner may inline into a particular routine. It limits the number of times that inlining can be applied to any routine.

If you specify `-no-inline-max-per-routine` (Linux and Mac OS X) or `/Qinline-max-per-routine-` (Windows), there is no limit to the number of times some routine may be inlined into a particular routine.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-size, Qinline-max-size

Specifies the lower limit for the size of what the inliner considers to be a large routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-size=`*n*

`-no-inline-max-size`

Windows:

`/Qinline-max-size=`*n*

`/Qinline-max-size-`

Arguments

n	Is a positive integer that specifies the minimum size of what the inliner considers to be a large routine.

Default

`-no-inline-max-size` or `/Qinline-max-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the lower limit for the size of what the inliner considers to be a large routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be medium and large-size routines.

The inliner prefers to inline small routines. It has a preference against inlining large routines. So, any large routine is highly unlikely to be inlined.

If you specify `-no-inline-max-size` (Linux and Mac OS X) or `/Qinline-max-size-` (Windows), there are no large routines. Every routine is either a small or medium routine.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-min-size`, `Qinline-min-size`
- `inline-factor`, `Qinline-factor`

- opt-report, Qopt-report
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-total-size, Qinline-max-total-size

Specifies how much larger a routine can normally grow when inline expansion is performed.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-total-size=n`

`-no-inline-max-total-size`

Windows:

`/Qinline-max-total-size=n`

`/Qinline-max-total-size-`

Arguments

n	Is a positive integer that specifies the permitted increase in the routine's size when inline expansion is performed.

Default

`-no-inline-max-total-size` or `/Qinline-max-total-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies how much larger a routine can normally grow when inline expansion is performed. It limits the potential size of the routine. For example, if 2000 is specified for *n*, the size of any routine will normally not increase by more than 2000.

If you specify `-no-inline-max-total-size` (Linux and Mac OS X) or `/Qinline-max-total-size-` (Windows), there is no limit to the size a routine may grow when inline expansion is performed.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-min-size, Qinline-min-size

Specifies the upper limit for the size of what the inliner considers to be a small routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-min-size=`*n*

`-no-inline-min-size`

Windows:

`/Qinline-min-size=`*n*

`/Qinline-min-size-`

Arguments

n	Is a positive integer that specifies the maximum size of what the inliner considers to be a small routine.

Default

`-no-inline-min-size` or `/Qinline-min-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the upper limit for the size of what the inliner considers to be a small routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be small and medium-size routines.

The inliner has a preference to inline small routines. So, when a routine is smaller than or equal to the specified size, it is very likely to be inlined.

If you specify `-no-inline-min-size` (Linux and Mac OS X) or `/Qinline-min-size-` (Windows), there is no limit to the size of small routines. Every routine is a small routine; there are no medium or large routines.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-min-size`, `Qinline-min-size`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

intconstant

Tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.

IDE Equivalent

Windows: **Compatibility > Use F77 Integer Constants**

Linux: None

Mac OS X: **Compatibility > Use F77 Integer Constants**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-intconstant
-nointconstant
```

Windows:

`/intconstant`

`/nointconstant`

Arguments

None

Default

`nointconstant`	The compiler uses the Fortran 95/90 default INTEGER type.

Description

This option tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.

With FORTRAN 77 semantics, the kind is determined by the value of the constant. All constants are kept internally by the compiler in the highest precision possible. For example, if you specify option `intconstant`, the compiler stores an integer constant of 14 internally as INTEGER(KIND=8) and converts the constant upon reference to the corresponding proper size. Fortran 95/90 specifies that integer constants with no explicit KIND are kept internally in the default INTEGER kind (KIND=4 by default).

Note that the internal precision for floating-point constants is controlled by option fpconstant.

Alternate Options

None

integer-size

Specifies the default KIND for integer and logical variables.

IDE Equivalent

Windows: **Data > Default Integer KIND**

Linux: None

Mac OS X: **Data > Default Integer KIND**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-integer-size` *size*

Windows:

`/integer-size:`*size*

Arguments

size	Is the size for integer and logical variables. Possible values are: 16, 32, or 64.

Default

`integer-size 32`	Integer and logical variables are 4 bytes long (INTEGER(KIND=4) and LOGICAL(KIND=4)).

Description

This option specifies the default size (in bits) for integer and logical variables.

Option	Description
`integer-size 16`	Makes default integer and logical variables 2 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=2).
`integer-size 32`	Makes default integer and logical variables 4 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=4).
`integer-size 64`	Makes default integer and logical variables 8 bytes long. INTEGER and LOGICAL declarations are treated as (KIND=8).

Alternate Options

`integer-size 16`	Linux and Mac OS X: `-i2` Windows: `/4I2`
`integer-size 32`	Linux and Mac OS X: `-i4`

	Windows: `/4I4`
`integer-size 64`	Linux and Mac OS X: `-i8` Windows: `/4I8`

ip, Qip

Determines whether additional interprocedural optimizations for single-file compilation are enabled.

IDE Equivalent

Windows: **Optimization > Interprocedural Optimization**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ip`

`-no-ip`

Windows:

`/Qip`

`/Qip-`

Arguments

None

Default

OFF	Some limited interprocedural optimizations occur, including inline function expansion for calls to functions defined within the current source file. These optimizations are a subset of full intra-file interprocedural optimizations. Note that this setting is not the same as `-no-ip` (Linux and Mac OS X) or `/Qip-` (Windows).

Description

This option determines whether additional interprocedural optimizations for single-file compilation are enabled.

Options `-ip` (Linux and Mac OS X) and `/Qip` (Windows) enable additional interprocedural optimizations for single-file compilation.

Options `-no-ip` (Linux and Mac OS X) and `/Qip-` (Windows) may not disable inlining. To ensure that inlining of user-defined functions is disabled, specify `-inline-level=0`or `-fno-inline` (Linux and Mac OS X), or specify `/Ob0` (Windows).

Alternate Options

None

See Also

-
-
- `finline-functions`

ip-no-inlining, Qip-no-inlining

Disables full and partial inlining enabled by interprocedural optimization options.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-ip-no-inlining
```

Windows:

```
/Qip-no-inlining
```

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables full and partial inlining enabled by the following interprocedural optimization options:

- On Linux and Mac OS X systems: `-ip` or `-ipo`
- On Windows systems: `/Qip`, `/Qipo`, or `/Ob2`

It has no effect on other interprocedural optimizations.

On Windows systems, this option also has no effect on user-directed inlining specified by option `/Ob1`.

Alternate Options

None

ip-no-pinlining, Qip-no-pinlining

Disables partial inlining enabled by interprocedural optimization options.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-ip-no-pinlining
```

Windows:

```
/Qip-no-pinlining
```

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables partial inlining enabled by the following interprocedural optimization options:

- On Linux and Mac OS X systems: `-ip` or `-ipo`
- On Windows systems: `/Qip` or `/Qipo`

It has no effect on other interprocedural optimizations.

Alternate Options

None

IPF-flt-eval-method0, QIPF-flt-eval-method0

Tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program. This is a deprecated option.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-IPF-flt-eval-method0
```

Mac OS X:

None

Windows:

/QIPF-flt-eval-method0

Arguments

None

Default

OFF	Expressions involving floating-point operands are evaluated by default rules.

Description

This option tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program.

By default, intermediate floating-point expressions are maintained in higher precision.

The recommended method to control the semantics of floating-point calculations is to use option -fp-model (Linux) or /fp (Windows).

Instead of using -IPF-flt-eval-method0 (Linux) or /QIPF-flt-eval-method0 (Windows), you can use -fp-model source (Linux) or /fp:source (Windows).

Alternate Options

None

See Also

-
-
- fp-model, fp

IPF-fltacc, QIPF-fltacc

Disables optimizations that affect floating-point accuracy. This is a deprecated option.

IDE Equivalent

Windows: **Floating Point > Floating-Point Accuracy**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

`-IPF-fltacc`

`-no-IPF-fltacc`

Mac OS X:

`None`

Windows:

`/QIPF-fltacc`

`/QIPF-fltacc-`

Arguments

None

Default

`-no-IPF-fltacc` or `/QIPF-fltacc-`	Optimizations are enabled that affect floating-point accuracy.

Description

This option disables optimizations that affect floating-point accuracy.

If the default setting is used, the compiler may apply optimizations that reduce floating-point accuracy.

You can use this option to improve floating-point accuracy, but at the cost of disabling some optimizations.

The recommended method to control the semantics of floating-point calculations is to use option `-fp-model` (Linux) or `/fp` (Windows).

Instead of using `-IPF-fltacc` (Linux) or `/QIPF-fltacc` (Windows), you can use `-fp-model precise` (Linux) or `/fp:precise` (Windows).

Instead of using `-no-IPF-fltacc` (Linux) or `/QIPF-fltacc-` (Windows), you can use `-fp-model fast` (Linux) or `/fp:fast` (Windows).

Alternate Options

None

See Also

-
-
- `fp-model, fp`

IPF-fma, QIPF-fma

See fma, Qfma.

IPF-fp-relaxed, QIPF-fp-relaxed

See fp-relaxed, Qfp-relaxed.

ipo, Qipo

Enables interprocedural optimization between files.

IDE Equivalent

Windows: **Optimization > Interprocedural Optimization**

General > Whole Program Optimization

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo[n]`

Windows:

`/Qipo[n]`

Arguments

n	Is an optional integer that specifies the number of object files the compiler should create. The integer must be greater than or equal to 0.

Default

OFF	Multifile interprocedural optimization is not enabled.

Description

This option enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or Whole Program Optimization (WPO).

When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

If *n* is 0, the compiler decides whether to create one or more object files based on an estimate of the size of the application. It generates one object file for small applications, and two or more object files for large applications.

If *n* is greater than 0, the compiler generates *n* object files, unless *n* exceeds the number of source files (*m*), in which case the compiler generates only *m* object files.

If you do not specify *n*, the default is 0.

Alternate Options

None

See Also

•

•

Optimizing Applications:

Interprocedural Optimization (IPO) Quick Reference

Interprocedural Optimization (IPO) Overview

Using IPO

ipo-c, Qipo-c

Tells the compiler to optimize across multiple files and generate a single object file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-c`

Windows:

`/Qipo-c`

Arguments

None

Default

OFF	The compiler does not generate a multifile object file.

Description

This option tells the compiler to optimize across multiple files and generate a single object file (named ipo_out.o on Linux and Mac OS X systems; ipo_out.obj on Windows systems).

It performs the same optimizations as `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), but compilation stops before the final link stage, leaving an optimized object file that can be used in further link steps.

Alternate Options

None

See Also

-
-
- `ipo`, `Qipo`

ipo-jobs, Qipo-jobs

Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-jobs`*n*

Windows:

`/Qipo-jobs:`*n*

Arguments

n	Is the number of commands (jobs) to run simultaneously. The number must be greater than or equal to 1.

Default

`-ipo-jobs1` or `/Qipo-jobs:1`	One command (job) is executed in an interprocedural optimization parallel build.

Description

This option specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). It should only be used if the link-time compilation is generating more than one object. In this case, each object is generated by a separate compilation, which can be done in parallel.

This option can be affected by the following compiler options:

- `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) when applications are large enough that the compiler decides to generate multiple object files.
- `-ipo`*n* (Linux and Mac OS X) or `/Qipo`*n* (Windows) when *n* is greater than 1.
- `-ipo-separate` (Linux) or `/Qipo-separate` (Windows)

CAUTION. Be careful when using this option. On a multi-processor system with lots of memory, it can speed application build time. However, if *n* is greater than the number of processors, or if there is not enough memory to avoid thrashing, this option can increase application build time.

Alternate Options

None

See Also

-
-
- `ipo`, `Qipo`
- `ipo-separate`, `Qipo-separate`

ipo-S, Qipo-S

Tells the compiler to optimize across multiple files and generate a single assembly file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-S`

Windows:

`/Qipo-S`

Arguments

None

Default

OFF	The compiler does not generate a multifile assembly file.

Description

This option tells the compiler to optimize across multiple files and generate a single assembly file (named ipo_out.s on Linux and Mac OS X systems; ipo_out.asm on Windows systems).

It performs the same optimizations as `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), but compilation stops before the final link stage, leaving an optimized assembly file that can be used in further link steps.

Alternate Options

None

See Also

•

-
- ipo, Qipo

ipo-separate, Qipo-separate

Tells the compiler to generate one object file for every source file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-ipo-separate`

Mac OS X:

`None`

Windows:

`/Qipo-separate`

Arguments

None

Default

OFF	The compiler decides whether to create one or more object files.

Description

This option tells the compiler to generate one object file for every source file. It overrides any `-ipo` (Linux) or `/Qipo` (Windows) specification.

Alternate Options

None

See Also

-
-
- ipo, Qipo

isystem

Specifies a directory to add to the start of the system include path.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-isystem`*dir*

Windows:

None

Arguments

dir	Is the directory to add to the system include path.

Default

OFF	The default system include path is used.

Description

This option specifies a directory to add to the system include path. The compiler searches the specified directory for include files after it searches all directories specified by the `-I` compiler option but before it searches the standard system directories. This option is provided for compatibility with gcc.

Alternate Options

None

ivdep-parallel, Qivdep-parallel

Tells the compiler that there is no loop-carried memory dependency in the loop following an IVDEP directive.

IDE Equivalent

Windows: **Optimization > IVDEP Directive Memory Dependency**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

```
-ivdep-parallel
```

Mac OS X:

```
None
```

Windows:

```
/Qivdep-parallel
```

Arguments

None

Default

OFF	There may be loop-carried memory dependency in a loop that follows an IVDEP directive.

Description

This option tells the compiler that there is no loop-carried memory dependency in the loop following an IVDEP There may be loop-carried memory dependency in a loop that follows an IVDEP directive.
This has the same effect as specifying the IVDEP:LOOP directive.

Alternate Options

None

See Also

-
-

Optimizing Applications: Absence of Loop-carried Memory Dependency with IVDEP Directive

l

Tells the linker to search for a specified library when linking.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-l`*`string`*

Windows:

`None`

Arguments

`string`	Specifies the library (lib*`string`*) that the linker should search.

Default

OFF	The linker searches for standard libraries in standard directories.

Description

This option tells the linker to search for a specified library when linking.

When resolving references, the linker normally searches for libraries in several standard directories, in directories specified by the `L` option, then in the library specified by the `l` option.

The linker searches and processes libraries and object files in the order they are specified. So, you should specify this option following the last object file it applies to.

Alternate Options

None

See Also

-
- `L`

L

Tells the linker to search for libraries in a specified directory before searching the standard directories.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-L`*`dir`*

Windows:

`None`

Arguments

dir	Is the name of the directory to search for libraries.

Default

OFF	The linker searches the standard directories for libraries.

Description

This option tells the linker to search for libraries in a specified directory before searching for them in the standard directories.

Alternate Options

None

See Also

-
- `l`

LD

See dll.

libdir

Controls whether linker options for search libraries are included in object files generated by the compiler.

IDE Equivalent

Windows: **Libraries > Disable Default Library Search Rules** (`/libdir:[no]automatic`)

Libraries > Disable OBJCOMMENT Library Name in Object (`/libdir:[no]user`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

None

Windows:

/libdir[:*keyword*]

/nolibdir

Arguments

keyword Specifies the linker search options. Possible values are:

none	Prevents any linker search options from being included into the object file. This is the same as specifying /nolibdir.
[no]automatic	Determines whether linker search options for libraries automatically determined by the ifort command driver (default libraries) are included in the object file.
[no]user	Determines whether linker search options for libraries specified by the OBJCOMMENT source directives are included in the object file.
all	Causes linker search options for the following libraries: • Libraries automatically determined by the ifort command driver (default libraries) • Libraries specified by the OBJCOMMENT directive to be included in the object file This is the same as specifying /libdir.

Default

`/libdir:all`	Linker search options for libraries automatically determined by the `ifort` command driver (default libraries) and libraries specified by the OBJCOMMENT directive are included in the object file.

Description

This option controls whether linker options for search libraries (`/DEFAULTLIB:library`) are included in object files generated by the compiler.

The linker option `/DEFAULTLIB:library` adds one library to the list of libraries that the linker searches when resolving references. A library specified with `/DEFAULTLIB:library` is searched after libraries specified on the command line and before default libraries named in .obj files.

Alternate Options

`/libdir:none`	Linux and Mac OS X: None Windows: `/Zl`

libs

Tells the compiler which type of run-time library to link to.

IDE Equivalent

Windows: **Libraries > Runtime Library** (`/libs:{static|dll|qwin|qwins}`, `/threads`, `/dbglibs`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/libs[:`*keyword*`]`

Arguments

keyword — Specifies the type of run-time library to link to. Possible values are:

`static`	Specifies a single-threaded, static library (same as specifying `/libs`).
`dll`	Specifies a single-threaded, dynamic-link (DLL) library.
`qwin`	Specifies the Fortran QuickWin library.
`qwins`	Specifies the Fortran Standard Graphics library.

Default

`/libs:static` or `/libs` — The compiler links to a single-threaded, static run-time library.

Description

This option tells the compiler which type of run-time library to link to.

The library can be statically or dynamically loaded, multithreaded (`/threads`) or single-threaded, or debug (`/dbglibs`) or nondebug.

If you use the `/libs:dll` option and an unresolved reference is found in the DLL, it gets resolved when the program is executed, during program loading, reducing executable program size.

If you use the `/libs:qwin` or `/libs:qwins` option with the `/dll` option, the compiler issues a warning.

You cannot use the `/libs:qwin` option and options `/libs:dll /threads`.

The following table shows which options to specify for different run-time libraries:

Type of Library	Options Required	Alternate Option
Single-threaded, static	`/libs:static` or `/libs` or `/static`	`/ML`

Type of Library	Options Required	Alternate Option
Multithreaded	/libs:static /threads	/MT
Debug single-threaded	/libs:static /dbglibs	/MLd
Debug multithreaded	/libs:static /threads /dbglibs	/MTd
Single-threaded, dynamic-link libraries (DLLs)	/libs:dll	/MDs
Debug single-threaded, dynamic-link libraries (DLLs)	/libs:dll /dbglibs	/MDsd
Multithreaded DLLs	/libs:dll /threads	/MD
Multithreaded debug DLLs	/libs:dll /threads /dbglibs	/MDd
Fortran QuickWin multi-doc applications	/libs:qwin	/MW
Fortran standard graphics (QuickWin single-doc) applications	/libs:qwins	/MWs
Debug Fortran QuickWin multi-doc applications	/libs:qwin /dbglibs	None
Debug Fortran standard graphics (QuickWin single-doc) applications	/libs:qwins /dbglibs	None

Alternate Options

`/libs:dll`	Linux and Mac OS X:None Windows: `/MDs`
`/libs:static`	Linux and Mac OS X: None Windows: `/ML`
`/libs:qwin`	Linux and Mac OS X: None Windows: `/MW`
`/libs:qwins`	Linux and Mac OS X: None Windows: `/MWs`

link

Passes user-specified options directly to the linker at compile time.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/link`

Arguments

None

Default

OFF	No user-specified options are passed directly to the linker.

Description

This option passes user-specified options directly to the linker at compile time.

All options that appear following `/link` are passed directly to the linker.

Alternate Options

None

See Also

-
- `Xlinker`

logo

Displays the compiler version information.

IDE Equivalent

Windows: **General > Suppress Startup Banner** (`/nologo`)

Linux: None

Mac OS X: **General > Show Startup Banner** (`-V`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-logo`

`-nologo`

Windows:

`/logo`

`/nologo`

Arguments

None

Default

Linux and Mac OS X: `nologo`	The compiler version information is not displayed.
Windows: `logo`	The compiler version information is displayed.

Description

This option displays the startup banner, which contains the following compiler version information:

- ID: unique identification number for the compiler
- x.y.z: version of the compiler
- years: years for which the software is copyrighted

This option can be placed anywhere on the command line.

Alternate Options

Linux and Mac OS X: `-V`

Windows: None

lowercase, Qlowercase

See names.

m

Tells the compiler to generate optimized code specialized for the processor that executes your program.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-m[processor]`

Windows:

`None`

Arguments

processor Indicates the processor for which code is generated. Possible values are:

`ia32`	Generates code that will run on any Pentium or later processor. Disables any default extended instruction settings, and any previously set extended instruction settings. This value is only available on Linux systems using IA-32 architecture.
`sse`	This is the same as specifying `ia32`.
`sse2`	Generates code for Intel® Streaming SIMD Extensions 2 (Intel® SSE2). This value is only available on Linux systems.
`sse3`	Generates code for Intel® Streaming SIMD Extensions 3 (Intel® SSE3).
`ssse3`	Generates code for Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3).
`sse4.1`	Generates code for Intel® Streaming SIMD Extensions 4 Vectorizing Compiler and Media Accelerators.

Default

Linux systems: `-msse2` Mac OS X systems using IA-32 architecture: `-msse3`	For more information on the default values, see Arguments above.

Mac OS X systems using
Intel® 64 architecture:
`-mssse3`

Description

This option tells the compiler to generate optimized code specialized for the processor that executes your program.

Code generated with the values ia32, sse, sse2, or sse3 should execute on any compatible non-Intel processor with support for the corresponding instruction set.

Options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Alternate Options

Linux and Mac OS X: None

Windows: `/arch`

See Also

-
- `x`, `Qx`
- `ax`, `Qax`
- `arch`

m32, m64

Tells the compiler to generate code for a specific architecture.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-m32`

`-m64`

Windows:

`None`

Arguments

None

Default

OFF	The compiler's behavior depends on the host system.

Description

These options tell the compiler to generate code for a specific architecture.

Option	Description
`-m32`	Tells the compiler to generate code for IA-32 architecture.
`-m64`	Tells the compiler to generate code for Intel® 64 architecture.

The `-m32` and `-m64` options are the same as Mac OS* X options `-arch i386` and `-arch x86_64`, respectively. Note that these options are provided for compatibility with gcc. They are not related to the Intel® Fortran compiler option `arch`.

Alternate Options

None

map

Tells the linker to generate a link map file.

IDE Equivalent

Windows: **Linker > Debug > Generate Map File**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/map[:file]`

`/nomap`

Arguments

file	Is the name for the link map file. It can be a file name or a directory name.

Default

`/nomap`	No link map is generated.

Description

This option tells the linker to generate a link map file.

Alternate Options

None

map-opts, Qmap-opts

Maps one or more compiler options to their equivalent on a different operating system.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-map-opts`

Mac OS X:

`None`

Windows:

`/Qmap-opts`

Arguments

None

Default

OFF	No platform mappings are performed.

Description

This option maps one or more compiler options to their equivalent on a different operating system. The result is output to `stdout`.

On Windows systems, the options you provide are presumed to be Windows options, so the options that are output to `stdout` will be Linux equivalents.
On Linux systems, the options you provide are presumed to be Linux options, so the options that are output to `stdout` will be Windows equivalents.

The tool can be invoked from the compiler command line or it can be used directly.

No compilation is performed when the option mapping tool is used.

This option is useful if you have both compilers and want to convert scripts or makefiles.

NOTE. Compiler options are mapped to their equivalent on the architecture you are using.
For example, if you are using a processor with IA-32 architecture, you will only see equivalent options that are available on processors with IA-32 architecture.

Alternate Options

None

Example

The following command line invokes the option mapping tool, which maps the Linux options to Windows-based options, and then outputs the results to `stdout`:

```
ifort -map-opts -xP -O2
```

The following command line invokes the option mapping tool, which maps the Windows options to Linux-based options, and then outputs the results to `stdout`:

```
ifort /Qmap-opts /QxP /O2
```

See Also

-
-

Building Applications: Using the Option Mapping Tool

march

Tells the compiler to generate code for a specified processor.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux:

`-march=`*`processor`*

Mac OS X:

`None`

Windows:

`None`

Arguments

processor	Is the processor for which the compiler should generate code. Possible values are:
`pentium3`	Generates code for Intel® Pentium® III processors.
`pentium4`	Generates code for Intel® Pentium® 4 processors.
`core2`	Generates code for the Intel® Core 2™ processor family.

Default

OFF or `-march=pentium4`	On IA-32 architecture, the compiler does not generate processor-specific code unless it is told to do so. On systems using Intel® 64 architecture, the compiler generates code for Intel Pentium 4 processors.

Description

This option tells the compiler to generate code for a specified processor.

Specifying `-march=pentium4` sets `-mtune=pentium4`.

For compatibility, a number of historical *processor* values are also supported, but the generated code will not differ from the default.

Alternate Options

None

mcmodel

Tells the compiler to use a specific memory model to generate code and store data.

IDE Equivalent

None

Architectures

Intel® 64 architecture

Syntax

Linux:

`-mcmodel=`*`mem_model`*

Mac OS X:

`None`

Windows:

`None`

Arguments

`mem_model` Is the memory model to use. Possible values are:

`small`	Tells the compiler to restrict code and data to the first 2GB of address space. All accesses of code and data can be done with Instruction Pointer (IP)-relative addressing.
`medium`	Tells the compiler to restrict code to the first 2GB; it places no memory restriction on data. Accesses of code can be done with IP-relative addressing, but accesses of data must be done with absolute addressing.
`large`	Places no memory restriction on code or data. All accesses of code and data must be done with absolute addressing.

Default

`-mcmodel=small` On systems using Intel® 64 architecture, the compiler restricts code and data to the first 2GB of address space. Instruction Pointer (IP)-relative addressing can be used to access code and data.

Description

This option tells the compiler to use a specific memory model to generate code and store data. It can affect code size and performance. If your program has COMMON blocks and local data with a total size smaller than 2GB, `-mcmodel=small` is sufficient. COMMONs larger than 2GB require `-mcmodel=medium` or `-mcmodel=large`. Allocation of memory larger than 2GB can be done with any setting of `-mcmodel`.

IP-relative addressing requires only 32 bits, whereas absolute addressing requires 64-bits. IP-relative addressing is somewhat faster. So, the `small` memory model has the least impact on performance.

NOTE. When you specify `-mcmodel=medium` or `-mcmodel=large`, you must also specify compiler option `-shared-intel` to ensure that the correct dynamic versions of the Intel run-time libraries are used.

Alternate Options

None

Example

The following example shows how to compile using `-mcmodel`:

```
ifort -shared-intel -mcmodel=medium -o  prog prog.f
```

See Also

-
- `shared-intel`
- `fpic`

mcpu

This is a deprecated option. See mtune.

MD

Tells the linker to search for unresolved references in a multithreaded, dynamic-link run-time library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/MD`

`/MDd`

Arguments

None

Default

OFF	The linker searches for unresolved references in a single-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a multithreaded, dynamic-link (DLL) run-time library. This is the same as specifying options `/libs:dll /threads /dbglibs`. You can also specify `/MDd`, where `d` indicates a debug version.

Alternate Options

None

See Also

-
- `libs`
- `threads`

MDs

Tells the linker to search for unresolved references in a single-threaded, dynamic-link run-time library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/MDs`

`/MDsd`

Arguments

None

Default

OFF	The linker searches for unresolved references in a single-threaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a single-threaded, dynamic-link (DLL) run-time library.

You can also specify `/MDsd`, where *d* indicates a debug version.

Alternate Options

`/MDs`	Linux and Mac OS X: None Windows: `/libs:dll`

See Also

-
- libs

mdynamic-no-pic

Generates code that is not position-independent but has position-independent external references.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux:

None

Mac OS X:

-mdynamic-no-pic

Windows:

None

Arguments

None

Default

OFF	All references are generated as position independent.

Description

This option generates code that is not position-independent but has position-independent external references.

The generated code is suitable for building executables, but it is not suitable for building shared libraries.

This option may reduce code size and produce more efficient code. It overrides the `-fpic` compiler option.

Alternate Options

None

See Also

-
- `fpic`

MG

See winapp.

mieee-fp

See fltconsistency.

minstruction, Qinstruction

Determines whether MOVBE instructions are generated for Intel processors.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-minstruction=[no]movbe
```

Windows:

```
/Qinstruction:[no]movbe
```

Arguments

None

Default

`-minstruction=nomovbe` or `/Qinstruction:nomovbe` The compiler does not generate MOVBE instructions for Intel® Atom™ processors.

Description

This option determines whether MOVBE instructions are generated for Intel processors. To use this option, you must also specify `-xSSE3_ATOM` (Linux and Mac OS X) or `/QxSSE3_ATOM` (Windows).

If `-minstruction=movbe` or `/Qinstruction:movbe` is specified, the following occurs:

- MOVBE instructions are generated that are specific to the Intel® Atom™ processor.
- The options are ON by default when `-xSSE3_ATOM` or `/QxSSE3_ATOM` is specified.
- Generated executables can only be run on Intel® Atom™ processors or processors that support Intel® Streaming SIMD Extensions 3 (Intel® SSE3) and MOVBE.

If `-minstruction=nomovbe` or `/Qinstruction:nomovbe` is specified, the following occurs:

- The compiler optimizes code for the Intel® Atom™ processor, but it does not generate MOVBE instructions.
- Generated executables can be run on non-Intel® Atom™ processors that support Intel® SSE3.

Alternate Options

None

See Also

-
-
- `x, Qx`

mixed-str-len-arg

See iface.

mkl, Qmkl

Tells the compiler to link to certain parts of the Intel® Math Kernel Library (Intel® MKL).

IDE Equivalent

Windows: **Libraries > Use Intel(R) Math Kernel Library**

Linux: None

Mac OS X: **Libraries > Use Inte(R)l Math Kernel Library**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-mkl[=lib]`

Windows:

`/Qmkl[:lib]`

Arguments

lib Indicates the part of the library that the compiler should link to. Possible values are:

`parallel`	Tells the compiler to link using the threaded part of the Intel® MKL. This is the default if the option is specified with no *lib*.
`sequential`	Tells the compiler to link using the non-threaded part of the Intel® MKL.
`cluster`	Tells the compiler to link using the cluster part and the sequential part of the Intel® MKL.

Default

OFF	The compiler does not link to the Intel® MKL.

Description

This option tells the compiler to link to certain parts of the Intel® Math Kernel Library (Intel® MKL).

Alternate Options

None

ML

Tells the linker to search for unresolved references in a single-threaded, static run-time library.
This option has been deprecated.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/ML`

`/MLd`

Arguments

None

Default

Systems using Microsoft Visual Studio 2003: `/ML`	If Microsoft* Visual Studio* 2003 is being used, the linker searches for unresolved references in a single-threaded, static run-time library.
Systems using Microsoft Visual Studio 2005 or later: OFF	If Microsoft* Visual Studio* 2005 or greater is being used, or Microsoft* Visual Studio* Premier Partner Edition (VSPPE) has been installed, the linker searches for unresolved references in a multithreaded, static run-time library.

Description

This option tells the linker to search for unresolved references in a single-threaded, static run-time library. It is only valid with Microsoft Visual Studio 2003, and is deprecated with later versions.

You can also specify `/MLd`, where `d` indicates a debug version.

Alternate Options

Linux: None

Mac OS X: None

Windows: `/libs:static`

See Also

-
- `libs`

module

Specifies the directory where module files should be placed when created and where they should be searched for.

IDE Equivalent

Windows: **Output > Module Path**

Linux: None

Mac OS X: **Output Files > Module Path**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-module` *path*

Windows:

`/module:`*path*

Arguments

path	Is the directory for module files.

Default

OFF	The compiler places module files in the current directory.

Description

This option specifies the directory (path) where module (.mod) files should be placed when created and where they should be searched for (USE statement).

Alternate Options

None

mp

See fltconsistency

multiple-processes, MP

Creates multiple processes that can be used to compile large numbers of source files at the same time.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-multiple-processes[=n]`

Windows:

`/MP[:n]`

Arguments

n	Is the maximum number of processes that the compiler should create.

Default

OFF	A single process is used to compile source files.

Description

This option creates multiple processes that can be used to compile large numbers of source files at the same time. It can improve performance by reducing the time it takes to compile source files on the command line.

This option causes the compiler to create one or more copies of itself, each in a separate process. These copies simultaneously compile the source files.

If *n* is not specified for this option, the default value is as follows:

- On Windows OS, the value is based on the setting of the NUMBER_OF_PROCESSORS environment variable.
- On Linux OS and Mac OS X, the value is 2.

This option applies to compilations, but not to linking or link-time code generation.

Alternate Options

None

mp1, Qprec

Improves floating-point precision and consistency.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-mp1`

Windows:

`/Qprec`

Arguments

None

Default

OFF	The compiler provides good accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option improves floating-point consistency. It ensures the out-of-range check of operands of transcendental functions and improves the accuracy of floating-point compares.

This option prevents the compiler from performing optimizations that change NaN comparison semantics and causes all values to be truncated to declared precision before they are used in comparisons. It also causes the compiler to use library routines that give better precision results compared to the X87 transcendental instructions.

This option disables fewer optimizations and has less impact on performance than option `flt-consistency` or `mp`.

Alternate Options

None

See Also

-
-
- `fltconsistency`
- `mp`

mrelax

Tells the compiler to pass linker option `-relax` *to the linker.*

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`-mrelax`

`-mno-relax`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

`-mno-relax`	The compiler does not pass `-relax` to the linker.

Description

This option tells the compiler to pass linker option `-relax` to the linker.

Alternate Options

None

MT

Tells the linker to search for unresolved references in a multithreaded, static run-time library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/MT`

`/MTd`

Arguments

None

Default

Systems using Intel® 64 architecture: `/MT /noreen-trancy`	On systems using Intel® 64 architecture, the linker searches for unresolved references in a multithreaded, static run-time library.
IA-32 architecture and IA-64 architecture: OFF	On systems using IA-32 architecture and IA-64 architecture, the linker searches for unresolved references in a single-threaded, static run-time library. However, on systems using IA-32 architecture, if option `Qvc8` is in effect, the linker searches for unresolved references in threaded libraries.

Description

This option tells the linker to search for unresolved references in a multithreaded, static run-time library. This is the same as specifying options `/libs:static /threads /noreentrancy`. You can also specify `/MTd`, where `d` indicates a debug version.

Alternate Options

None

See Also

-
- `Qvc`
- `libs`
- `threads`
- `reentrancy`

mtune

Performs optimizations for specific processors.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-mtune=`*`processor`*

Windows:

`None`

Arguments

`processor`	Is the processor for which the compiler should perform optimizations. Possible values are:

`generic`	Generates code for the compiler's default behavior.
`core2`	Optimizes for the Intel® Core™ 2 processor family, including support for MMX™, Intel® SSE, SSE2, SSE3 and SSSE3 instruction sets.
`pentium`	Optimizes for Intel® Pentium® processors.
`pentium-mmx`	Optimizes for Intel® Pentium® with MMX technology.
`pentiumpro`	Optimizes for Intel® Pentium® Pro, Intel Pentium II, and Intel Pentium III processors.
`pentium4`	Optimizes for Intel® Pentium® 4 processors.
`pentium4m`	Optimizes for Intel® Pentium® 4 processors with MMX technology.
`itanium2`	Optimizes for Intel® Itanium® 2 processors.
`itanium2-p9000`	Optimizes for the Dual-Core Intel® Itanium® 2 processor 9000 series. This option affects the order of the generated instructions, but the generated instructions are limited to Intel® Itanium® 2 processor instructions unless the program uses (executes) intrinsics specific to the Dual-Core Intel® Itanium® 2 processor 9000 series.

Default

generic	On systems using IA-32 and Intel® 64 architectures, code is generated for the compiler's default behavior.
itanium2-p9000	On systems using IA-64 architecture, the compiler optimizes for the Dual-Core Intel® Itanium® 2 processor 9000 series.

Description

This option performs optimizations for specific processors.

The resulting executable is backwards compatible and generated code is optimized for specific processors. For example, code generated with `-mtune=itanium2-p9000` will run correctly on single-core Itanium® 2 processors, but it might not run as fast as if it had been generated using `-mtune=itanium2`.

The following table shows on which architecture you can use each value.

	Architecture		
processor **Value**	**IA-32 architecture**	**Intel® 64 architecture**	**IA-64 architecture**
`generic`	X	X	X
`core2`	X	X	
`pentium`	X		
`pentium-mmx`	X		
`pentiumpro`	X		
`pentium4`	X		
`pentium4m`	X		
`itanium2`			X
`itanium2-p9000`			X

Alternate Options

`-mtune`	Linux: `-mcpu` (this is a deprecated option) Mac OS X: None Windows: None
`-mtune=itanium2`	Linux: `-mcpu=itanium2` (`-mcpu` is a deprecated option) Mac OS X: None Windows: `/G2`
`-mtune=itanium2-p9000`	Linux: `-mcpu=itanium2-p9000` (`-mcpu` is a deprecated option) Mac OS X: None Windows: `/G2-p9000`

multiple-processes, MP

Creates multiple processes that can be used to compile large numbers of source files at the same time.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-multiple-processes[=n]`

Windows:

`/MP[:n]`

Arguments

n	Is the maximum number of processes that the compiler should create.

Default

OFF	A single process is used to compile source files.

Description

This option creates multiple processes that can be used to compile large numbers of source files at the same time. It can improve performance by reducing the time it takes to compile source files on the command line.

This option causes the compiler to create one or more copies of itself, each in a separate process. These copies simultaneously compile the source files.

If *n* is not specified for this option, the default value is as follows:

- On Windows OS, the value is based on the setting of the NUMBER_OF_PROCESSORS environment variable.

- On Linux OS and Mac OS X, the value is 2.

This option applies to compilations, but not to linking or link-time code generation.

Alternate Options

None

MW

See libs.

MWs

See libs.

names

Specifies how source code identifiers and external names are interpreted.

IDE Equivalent

Windows: **External Procedures > Name Case Interpretation**

Linux: None

Mac OS X: **External Procedures > Name Case Interpretation**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-names keyword`

Windows:

`/names:keyword`

Arguments

keyword Specifies how to interpret the identifiers and external names in source code. Possible values are:

`lowercase`	Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase.
`uppercase`	Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase.
`as_is`	Causes the compiler to distinguish case differences in identifiers and to preserve the case of external names.

Default

`lowercase`	This is the default on Linux and Mac OS X systems. The compiler ignores case differences in identifiers and converts external names to lowercase.
`uppercase`	This is the default on Windows systems. The compiler ignores case differences in identifiers and converts external names to uppercase.

Description

This option specifies how source code identifiers and external names are interpreted. It can be useful in mixed-language programming.

This naming convention applies whether names are being defined or referenced.

You can use the ALIAS directive to specify an alternate external name to be used when referring to external subprograms.

CAUTION. On Windows systems, if you specify option `/iface:cref`, it overrides the default for external names and causes them to be lowercase. It is as if you specified "!dec$ attributes c, reference" for the external name.

If you specify option `/iface:cref` and want external names to be uppercase, you must explicitly specify option `/names:uppercase`.

Alternate Options

`names lowercase`	Linux and Mac OS X: `-lowercase` Windows: `/Qlowercase`
`names uppercase`	Linux and Mac OS X: `-uppercase` Windows: `/Quppercase`

See Also

-
- `iface`
- ALIAS Directive

nbs

See assume.

no-bss-init, Qnobss-init

Tells the compiler to place in the DATA section any variables explicitly initialized with zeros.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-no-bss-init
```

Windows:

```
/Qnobss-init
```

Arguments

None

Default

OFF	Variables explicitly initialized with zeros are placed in the BSS section.

Description

This option tells the compiler to place in the DATA section any variables explicitly initialized with zeros.

Alternate Options

Linux and Mac OS X: `-nobss-init` (this is a deprecated option)

Windows: None

nodefaultlibs

Prevents the compiler from using standard libraries when linking.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-nodefaultlibs`

Windows:

`None`

Arguments

None

Default

OFF	The standard libraries are linked.

Description

This option prevents the compiler from using standard libraries when linking.

Alternate Options

None

See Also

-
- nostdlib

nodefine

See D.

nofor-main

Specifies that the main program is not written in Fortran.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-nofor-main
```

Windows:

```
None
```

Arguments

None

Default

OFF	The compiler assumes the main program is written in Fortran.

Description

This option specifies that the main program is not written in Fortran. It is a link-time option that prevents the compiler from linking `for_main.o` into applications.

For example, if the main program is written in C and calls a Fortran subprogram, specify `-nofor-main` when compiling the program with the `ifort` command.

If you omit this option, the main program must be a Fortran program.

Alternate Options

None

noinclude

See X.

nolib-inline

Disables inline expansion of standard library or intrinsic functions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-nolib-inline`

Windows:

`None`

Arguments

None

Default

OFF	The compiler inlines many standard library and intrinsic functions.

Description

This option disables inline expansion of standard library or intrinsic functions. It prevents the unexpected results that can arise from inline expansion of these functions.

Alternate Options

None

nostartfiles

Prevents the compiler from using standard startup files when linking.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-nostartfiles
```

Windows:

```
None
```

Arguments

None

Default

OFF	The compiler uses standard startup files when linking.

Description

This option prevents the compiler from using standard startup files when linking.

Alternate Options

None

See Also

-
- nostdlib

nostdinc

See X.

nostdlib

Prevents the compiler from using standard libraries and startup files when linking.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-nostdlib`

Windows:

`None`

Arguments

None

Default

OFF	The compiler uses standard startup files and standard libraries when linking.

Description

This option prevents the compiler from using standard libraries and startup files when linking.

Alternate Options

None

See Also

-
- nodefaultlibs
- nostartfiles

nus

See assume.

o

Specifies the name for an output file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ofile`

Windows:

`None`

Arguments

file	Is the name for the output file. The space before *file* is optional.

Default

OFF	The compiler uses the default file name for an output file.

Description

This option specifies the name for an output file as follows:

- If `-c` is specified, it specifies the name of the generated object file.
- If `-S` is specified, it specifies the name of the generated assembly listing file.
- If -preprocess-only or `-P` is specified, it specifies the name of the generated preprocessor file.

Otherwise, it specifies the name of the executable file.

NOTE. If you misspell a compiler option beginning with "o", such as `-openmp`, `-opt-report`, etc., the compiler interprets the misspelled option as an `-o` *`file`* option. For example, say you misspell "`-opt-report`" as " `-opt-reprt`"; in this case, the compiler interprets the misspelled option as "`-o` pt-reprt", where pt-reprt is the output file name.

Alternate Options

Linux and Mac OS X: None

Windows: `/Fe`, `/exe`

See Also

-
- `Fe`
- `object`

O

Specifies the code optimization for applications.

IDE Equivalent

Windows: **General > Optimization** (`/Od`, `/O1`, `/O2`, `/O3`, `/fast`)

Optimization > Optimization (`/Od`, `/O1`, `/O2`, `/O3`, `/fast`)

Linux: None

Mac OS X: **General > Optimization Level** (`-O`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-O[n]`

Windows:

`/O[n]`

Arguments

n	Is the optimization level. Possible values are 1, 2, or 3. On Linux and Mac OS X systems, you can also specify 0.

Default

`O2`	Optimizes for code speed. This default may change depending on which other compiler options are specified. For details, see below.

Description

This option specifies the code optimization for applications.

Option	Description
`O` (Linux and Mac OS X)	This is the same as specifying `O2`.
`O0` (Linux and Mac OS X)	Disables all optimizations. On systems using IA-32 architecture and Intel® 64 architecture, this option sets option `-fno-omit-frame-pointer` and option `-fmath-errno`. This option causes certain `warn` options to be ignored. This is the default if you specify option `-debug` (with no keyword).
`O1`	Enables optimizations for speed and disables some optimizations that increase code size and affect speed. To limit code size, this option: • Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling. • On systems using IA-64 architecture, it disables software pipelining, loop unrolling, and global code scheduling.

Option	Description
	On systems using IA-64 architecture, this option also enables optimizations for server applications (straight-line and branch-like code with a flat profile). The `O1` option sets the following options: • On Linux and Mac OS X systems: `-funroll-loops0`, `-nofltconsistency` (same as `-mno-ieee-fp`), `-fomit-frame-pointer`, `-ftz` • On Windows systems using IA-32 architecture: `/Qunroll0`, `/nofltconsistency` (same as `/Op-`), `/Oy`, `/Os`, `/Ob2`, `/Qftz` • On Windows systems using Intel® 64 architecture and IA-64 architecture: `/Qunroll0`, `/nofltconsistency` (same as `/Op-`), `/Os`, `/Ob2`, `/Qftz` The O1 option may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

`O2`	Enables optimizations for speed. This is the generally recommended optimization level.
	Vectorization is enabled at O2 and higher levels.
	On systems using IA-64 architecture, this option enables optimizations for speed, including global code scheduling, software pipelining, predication, and speculation.
	This option also enables:
	• Inlining of intrinsics
	• Intra-file interprocedural optimization, which includes:
	  • inlining
	  • constant propagation
	  • forward substitution
	  • routine attribute propagation
	  • variable address-taken analysis
	  • dead static function elimination

Option	Description
	• removal of unreferenced variables • The following capabilities for performance gain: • constant propagation • copy propagation • dead-code elimination • global register allocation • global instruction scheduling and control speculation • loop unrolling • optimized code selection • partial redundancy elimination • strength reduction/induction variable simplification • variable renaming • exception handling optimizations • tail recursions • peephole optimizations • structure assignment lowering and optimizations • dead store elimination On Windows systems, this option is the same as the `Ox` option. The `O2` option sets the following options: • On Windows systems using IA-32 architecture: `/Og`, `/Ot`, `/Oy`, `/Ob2`, `/Gs`, and `/Qftz` • On Windows systems using Intel® 64 architecture: `/Og`, `/Ot`, `/Ob2`, `/Gs`, and `/Qftz` On Linux and Mac OS X systems, if `-g` is specified, `O2` is turned off and `O0` is the default unless `O2` (or `O1` or `O3`) is explicitly specified in the command line together with `-g`.

Option	Description
	This option sets other options that optimize for code speed. The options set are determined by the compiler depending on which architecture and operating system you are using.
O3	Enables O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations. Enables optimizations for maximum speed, such as: • Loop unrolling, including instruction scheduling • Code replication to eliminate branches • Padding the size of certain power-of-two arrays to allow more efficient cache use. On Windows systems, the O3 option sets the /Ob2 option. On Linux and Mac OS X systems, the O3 option sets option -fomit-frame-pointer. On systems using IA-32 architecture or Intel® 64 architecture, when O3 is used with options -ax or -x (Linux) or with options /Qax or /Qx (Windows), the compiler performs more aggressive data dependency analysis than for O2, which may result in longer compilation times. On systems using IA-64 architecture, the O3 option enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch. The O3 optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to O2 optimizations. The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets.

The last O option specified on the command line takes precedence over any others.

NOTE. The options set by the O option may change from release to release.

Alternate Options

O1	Linux and Mac OS X: None Windows: /Od, /optimize:0, /nooptimize
O2	Linux and Mac OS X: None Windows: /optimize:1, /optimize:2
O3	Linux and Mac OS X: None Windows: /Ox, /optimize:3, /optimize:4
O4	Linux and Mac OS X: None Windows: /optimize:5

See Also

-
- Od
- Op
- fast

Optimizing Applications:

Compiler Optimizations Overview

Optimization Options Summary

Efficient Compilation

inline-level, Ob

Specifies the level of inline function expansion.

IDE Equivalent

Windows: **Optimization > Inline Function Expansion**

Linux: None

Mac OS X: **Optimization > Inline Function Expansion**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-level=`*n*

Windows:

`/Ob`*n*

Arguments

n	Is the inline function expansion level. Possible values are 0, 1, and 2.

Default

`-inline-level=2` or `/Ob2`	This is the default if option `O2` is specified or is in effect by default. On Windows systems, this is also the default if option `O3` is specified.
`-inline-level=0` or `/Ob0`	This is the default if option `-O0` (Linux and Mac OS) or `/Od` (Windows) is specified.

Description

This option specifies the level of inline function expansion. Inlining procedures can greatly improve the run-time performance of certain programs.

Option	Description
`-inline-level=0` or `Ob0`	Disables inlining of user-defined functions. Note that statement functions are always inlined.
`-inline-level=1` or `Ob1`	Enables inlining when an inline keyword or an inline directive is specified.
`-inline-level=2` or `Ob2`	Enables inlining of any function at the compiler's discretion.

Alternate Options

Linux: `-Ob` (this is a deprecated option)

Mac OS X: None

Windows: None

See Also

-
-
- `inline`

object

Specifies the name for an object file.

IDE Equivalent

Windows: **Output Files > Object File Name**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
None
```

Windows:

```
/object:file
```

Arguments

file	Is the name for the object file. It can be a file or directory name. A directory name must be followed by a backslash (\). If a special character appears within the file name or directory name, the file name or directory name must appear within quotes. To be safe, you should consider any non-ASCII numeric character to be a special character.

Default

OFF	An object file has the same name as the name of the first source file and a file extension of .obj.

Description

This option specifies the name for an object file.

If you specify this option and you omit `/c` or `/compile-only`, the `/object` option gives the object file its name.

On Linux and Mac OS X systems, this option is equivalent to specifying option `-o`*`file`* `-c`.

Alternate Options

Linux and Mac OS X: None

Windows: `/Fo`

Example

The following command shows how to specify a directory:

```
ifort /object:directorya\ end.f
```

If you do not add the backslash following a directory name, an executable is created. For example, the following command causes the compiler to create directorya.exe:

```
ifort /object:directorya end.f
```

The following commands show how to specify a subdirectory that contains a special character:

```
ifort /object:"blank subdirectory"\ end.f
```

```
ifort /object:"c:\my_directory"\ end.f
```

See Also

-
- ○

Od

Disables all optimizations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Od`

Arguments

None

Default

OFF	The compiler performs default optimizations.

Description

This option disables all optimizations. It can be used for selective optimizations, such as a combination of `/Od` and `/Og` (disables all optimizations except global optimizations), or `/Od` and `/Ob1` (disables all optimizations, but enables inlining).

This option also causes certain `/warn` options to be ignored.

On IA-32 architecture, this option sets the `/Oy-` option.

Alternate Options

Linux and Mac OS X: `-O0`

Windows: /optimize:0

See Also

-
- `O`

Og

Enables global optimizations.

IDE Equivalent

Windows: **Optimization > Global Optimizations**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Og`

`/Og-`

Arguments

None

Default

`/Og`	Global optimizations are enabled unless `/Od` is specified.

Description

This option enables global optimizations.

Alternate Options

None

onetrip, Qonetrip

Tells the compiler to follow the FORTRAN 66 Standard and execute DO loops at least once.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-onetrip`

Windows:

`/Qonetrip`

Arguments

None

Default

OFF	The compiler applies the current Fortran Standard semantics, which allows zero-trip DO loops.

Description

This option tells the compiler to follow the FORTRAN 66 Standard and execute DO loops at least once.

Alternate Options

Linux and Mac OS X: -1

Windows: /1

Op

This is a deprecated option. See fltconsistency.

openmp, Qopenmp

Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.*

IDE Equivalent

Windows: **Language > Process OpenMP Directives**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp`

Windows:

`/Qopenmp`

Arguments

None

Default

OFF	No OpenMP multi-threaded code is generated by the compiler.

Description

This option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

If you use this option, multithreaded libraries are used, but option `fpp` is not automatically invoked.

This option sets option `automatic`.

This option works with any optimization level. Specifying no optimization (`-O0` on Linux or `/Od` on Windows) helps to debug OpenMP applications.

NOTE. On Mac OS X systems, when you enable OpenMP*, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode or an error will be displayed.

Alternate Options

None

See Also

-
-
- `openmp-stubs`, `Qopenmp-stubs`

openmp-lib, Qopenmp-lib

Lets you specify an OpenMP run-time library to use for linking.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-lib` *type*

Mac OS X:

`None`

Windows:

`/Qopenmp-lib:`*type*

Arguments

type	Specifies the type of library to use; it implies compatibility levels. Possible values are:
`legacy`	Tells the compiler to use the legacy OpenMP* run-time library (libguide). This setting does not provide compatibility with object files created using other compilers. This is a deprecated option.
`compat`	Tells the compiler to use the compatibility OpenMP* run-time library (libiomp). This setting provides compatibility with object files created using Microsoft* and GNU* compilers.

Default

`-openmp-lib compat` or `/Qopenmp-lib:compat`	The compiler uses the compatibility OpenMP* run-time library (libiomp).

Description

This option lets you specify an OpenMP* run-time library to use for linking.

The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

The compatibility OpenMP run-time library is compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) and GNU OpenMP run-time library (libgomp).

To use the compatibility OpenMP run-time library, compile and link your application using the `-openmp-lib compat` (Linux) or `/Qopenmp-lib:compat` (Windows) option. To use this option, you must also specify one of the following compiler options:

- Linux OS: `-openmp`, `-openmp-profile`, or `-openmp-stubs`
- Windows OS: `/Qopenmp`, `/Qopenmp-profile`, or `/Qopenmp-stubs`

On Windows* systems, the compatibility OpenMP* run-time library lets you combine OpenMP* object files compiled with the Microsoft* C/C++ compiler with OpenMP* object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

On Linux* systems, the compatibility Intel OpenMP* run-time library lets you combine OpenMP* object files compiled with the GNU* gcc or gfortran compilers with similar OpenMP* object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

You cannot link object files generated by the Intel® Fortran compiler to object files compiled by the GNU Fortran compiler, regardless of the presence or absence of the `-openmp` (Linux) or `/Qopenmp` (Windows) compiler option. This is because the Fortran run-time libraries are incompatible.

NOTE. The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compiler earlier than 10.0.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`
- `openmp-stubs`, `Qopenmp-stubs`
- `openmp-profile`, `Qopenmp-profile`

openmp-link, Qopenmp-link

Controls whether the compiler links to static or dynamic OpenMP run-time libraries.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-link` *`library`*

Windows:

`/Qopenmp-link:`*`library`*

Arguments

`library`	Specifies the OpenMP library to use. Possible values are:	
	`static`	Tells the compiler to link to static OpenMP run-time libraries.
	`dynamic`	Tells the compiler to link to dynamic OpenMP run-time libraries.

Default

`-openmp-link dynamic` or `/Qopenmp-link:dynamic`	The compiler links to dynamic OpenMP run-time libraries. However, if option `static` is specified, the compiler links to static OpenMP run-time libraries.

Description

This option controls whether the compiler links to static or dynamic OpenMP run-time libraries.

To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-openmp-link static` (Linux and Mac OS X) or `/Qopenmp-link:static` (Windows). However, we strongly recommend you use the default setting, `-openmp-link dynamic` (Linux and Mac OS X) or `/Qopenmp-link:dynamic` (Windows).

NOTE. Compiler options `-static-intel` and `-shared-intel` (Linux and Mac OS X) have no effect on which OpenMP run-time library is linked.

Alternate Options

None

openmp-profile, Qopenmp-profile

Enables analysis of OpenMP applications if Intel® Thread Profiler is installed.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-profile`

Mac OS X:

`None`

Windows:

`/Qopenmp-profile`

Arguments

None

Default

OFF	OpenMP applications are not analyzed.

Description

This option enables analysis of OpenMP* applications. To use this option, you must have previously installed Intel® Thread Profiler, which is one of the Intel® Threading Analysis Tools.

This option can adversely affect performance because of the additional profiling and error checking invoked to enable compatibility with the threading tools. Do not use this option unless you plan to use the Intel® Thread Profiler.

For more information about Intel® Thread Profiler, open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

openmp-report, Qopenmp-report

Controls the OpenMP parallelizer's level of diagnostic messages.*

IDE Equivalent

Windows: **Compilation Diagnostics > OpenMP Diagnostic Level**

Linux: None

Mac OS X: **Compiler Diagnostics > OpenMP Report**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-report[n]`

Windows:

`/Qopenmp-report[n]`

Arguments

n	Is the level of diagnostic messages to display. Possible values are:	
	`0`	No diagnostic messages are displayed.
	`1`	Diagnostic messages are displayed indicating loops, regions, and sections successfully parallelized.
	`2`	The same diagnostic messages are displayed as specified by openmp_report1 plus diagnostic messages indicating successful handling of MASTER constructs,

SINGLE constructs, CRITICAL constructs, ORDERED constructs, ATOMIC directives, and so forth.

Default

`-openmp-report1` or `/Qopenmp-report1`	If you do not specify *n*, the compiler displays diagnostic messages indicating loops, regions, and sections successfully parallelized. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the OpenMP* parallelizer's level of diagnostic messages. To use this option, you must also specify `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows).

If this option is specified on the command line, the report is sent to `stdout`.

On Windows systems, if this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`

Optimizing Applications:

Using Parallelism

OpenMP* Report

openmp-stubs, Qopenmp-stubs

Enables compilation of OpenMP programs in sequential mode.

IDE Equivalent

Windows: **Language > Process OpenMP Directives**

Linux: None

Mac OS X: **Language > Process OpenMP Directives**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-stubs`

Windows:

`/Qopenmp-stubs`

Arguments

None

Default

OFF	The library of OpenMP function stubs is not linked.

Description

This option enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`

openmp-threadprivate, Qopenmp-threadprivate

Lets you specify an OpenMP threadprivate implementation.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-threadprivate` *type*

Mac OS X:

`None`

Windows:

`/Qopenmp-threadprivate:`*type*

Arguments

type Specifies the type of threadprivate implementation. Possible values are:

`legacy` Tells the compiler to use the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler. This setting does not provide compatibility with the implementation used by other compilers.

`compat` Tells the compiler to use the compatibility OpenMP* threadprivate implementation based on applying the thread-local attribute to each threadprivate variable. This setting provides compatibility with the implementation provided by the Microsoft* and GNU* compilers.

Default

`-openmp-threadprivate legacy` or `/Qopenmp-threadprivate:legacy`	The compiler uses the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler.

Description

This option lets you specify an OpenMP* threadprivate implementation.

The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

To use this option, you must also specify one of the following compiler options:

- Linux OS: `-openmp`, `-openmp-profile`, or `-openmp-stubs`
- Windows OS: `/Qopenmp`, `/Qopenmp-profile`, or `/Qopenmp-stubs`

The value specified for this option is independent of the value used for option `-openmp-lib` (Linux) or `/Qopenmp-lib` (Windows).

Alternate Options

None

opt-block-factor, Qopt-block-factor

Lets you specify a loop blocking factor.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-block-factor=`*n*

Windows:

`/Qopt-block-factor:`*n*

Arguments

n	Is the blocking factor. It must be an integer. The compiler may ignore the blocking factor if the value is 0 or 1.

Default

OFF	The compiler uses default heuristics for loop blocking.

Description

This option lets you specify a loop blocking factor.

Alternate Options

None

opt-jump-tables, Qopt-jump-tables

Enables or disables generation of jump tables for switch statements.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-jump-tables=`*`keyword`*

`-no-opt-jump-tables`

Windows:

`/Qopt-jump-tables:`*`keyword`*

`/Qopt-jump-tables-`

Arguments

keyword	Is the instruction for generating jump tables. Possible values are:	
	`never`	Tells the compiler to never generate jump tables. All switch statements are implemented as chains of if-then-elses. This is the same as specifying `-no-opt-jump-tables` (Linux and Mac OS) or `/Qopt-jump-tables-` (Windows).
	`default`	The compiler uses default heuristics to determine when to generate jump tables.
	`large`	Tells the compiler to generate jump tables up to a certain pre-defined size (64K entries).
	`n`	Must be an integer. Tells the compiler to generate jump tables up to *n* entries in size.

Default

`-opt-jump-tables=default` or `/Qopt-jump-tables:default`	The compiler uses default heuristics to determine when to generate jump tables for switch statements.

Description

This option enables or disables generation of jump tables for switch statements. When the option is enabled, it may improve performance for programs with large switch statements.

Alternate Options

None

opt-loadpair, Qopt-loadpair

Enables or disables loadpair optimization.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-loadpair`

`-no-opt-loadpair`

Mac OS X:

`None`

Windows:

`/Qopt-loadpair`

`/Qopt-loadpair-`

Arguments

None

Default

`-no-opt-loadpair` or `/Qopt-loadpair-`	Loadpair optimization is disabled unless option `O3` is specified.

Description

This option enables or disables loadpair optimization.

When `-O3` is specified on IA-64 architecture, loadpair optimization is enabled by default. To disable loadpair generation, specify `-no-opt-loadpair` (Linux) or `/Qopt-loadpair-` (Windows).

Alternate Options

None

opt-malloc-options

Lets you specify an alternate algorithm for malloc().

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-malloc-options=`*n*

Windows:

`None`

Arguments

n Specifies the algorithm to use for malloc(). Possible values are:

`0`	Tells the compiler to use the default algorithm for malloc(). This is the default.
`1`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x10000000.
`2`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=2 and M_TRIM_THRESHOLD=0x40000000.
`3`	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0 and M_TRIM_THRESHOLD=-1.

4	Causes the following adjustments to the malloc() algorithm: M_MMAP_MAX=0, M_TRIM_THRESHOLD=-1, M_TOP_PAD=4096.

Default

-opt-malloc-options=0	The compiler uses the default algorithm when malloc() is called. No call is made to mallopt().

Description

This option lets you specify an alternate algorithm for malloc().

If you specify a non-zero value for *n*, it causes alternate configuration parameters to be set for how malloc() allocates and frees memory. It tells the compiler to insert calls to mallopt() to adjust these parameters to malloc() for dynamic memory allocation. This may improve speed.

Alternate Options

None

See Also

•

malloc(3) man page

mallopt function (defined in malloc.h)

opt-mem-bandwidth, Qopt-mem-bandwidth

Enables performance tuning and heuristics that control memory bandwidth use among processors.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-mem-bandwidth`*n*

Mac OS X:

`None`

Windows:

`/Qopt-mem-bandwidth`*n*

Arguments

n	Is the level of optimizing for memory bandwidth usage. Possible values are:	
	`0`	Enables a set of performance tuning and heuristics in compiler optimizations that is optimal for serial code.
	`1`	Enables a set of performance tuning and heuristics in compiler optimizations for multithreaded code generated by the compiler.
	`2`	Enables a set of performance tuning and heuristics in compiler optimizations for parallel code such as Windows Threads, pthreads, and MPI code, besides multithreaded code generated by the compiler.

Default

`-opt-mem-bandwidth0` or `/Qopt-mem-bandwidth0`	For serial (non-parallel) compilation, a set of performance tuning and heuristics in compiler optimizations is enabled that is optimal for serial code.
`-opt-mem-bandwidth1` or `/Qopt-mem-bandwidth1`	If you specify compiler option `-parallel` (Linux) or `/Qparallel` (Windows), or `-openmp` (Linux) or `/Qopenmp` (Windows), a set of performance tuning and heuristics in compiler optimizations for multithreaded code generated by the compiler is enabled.

Description

This option enables performance tuning and heuristics that control memory bandwidth use among processors. It allows the compiler to be less aggressive with optimizations that might consume more bandwidth, so that the bandwidth can be well-shared among multiple processors for a parallel program.

For values of *n* greater than 0, the option tells the compiler to enable a set of performance tuning and heuristics in compiler optimizations such as prefetching, privatization, aggressive code motion, and so forth, for reducing memory bandwidth pressure and balancing memory bandwidth traffic among threads.

This option can improve performance for threaded or parallel applications on multiprocessors or multicore processors, especially when the applications are bounded by memory bandwidth.

Alternate Options

None

See Also

-
-
- `parallel`, `Qparallel`
- `openmp`, `Qopenmp`

opt-mod-versioning, Qopt-mod-versioning

Enables or disables versioning of modulo operations for certain types of operands.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-opt-mod-versioning
-no-opt-mod-versioning
```

Mac OS X:

None

Windows:

/Qopt-mod-versioning

/Qopt-mod-versioning-

Arguments

None

Default

-no-opt-mod-versioning Versioning of modulo operations is disabled.
or/Qopt-mod-versioning-

Description

This option enables or disables versioning of modulo operations for certain types of operands. It is used for optimization tuning.

Versioning of modulo operations may improve performance for x mod y when modulus y is a power of 2.

Alternate Options

None

opt-multi-version-aggressive, Qopt-multi-version-aggressive

Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-multi-version-aggressive`

`-no-opt-multi-version-aggressive`

Windows:

`/Qopt-multi-version-aggressive`

`/Qopt-multi-version-aggressive-`

Arguments

None

Default

`-no-opt-multi-version-aggressive` or `/Qopt-multi-version-aggressive-`	The compiler uses default heuristics when checking for pointer aliasing and scalar replacement.

Description

This option tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement. This option may improve performance.

Alternate Options

None

opt-prefetch, Qopt-prefetch

Enables or disables prefetch insertion optimization.

IDE Equivalent

Windows: **Optimization > Prefetch Insertion**

Linux: None

Mac OS X: **Optimization > Enable Prefetch Insertion**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-prefetch[=n]`

`-no-opt-prefetch`

Windows:

`/Qopt-prefetch[:n]`

`/Qopt-prefetch-`

Arguments

n	Is the level of detail in the report. Possible values are:	
	`0`	Disables software prefetching. This is the same as specifying `-no-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch-` (Windows).
	`1` to `4`	Enables different levels of software prefetching. If you do not specify a value for *n*, the default is `2` on IA-32 and Intel® 64 architecture; the default is `3` on IA-64 architecture. Use lower values to reduce the amount of prefetching.

Default

IA-64 architecture: `-opt-prefetch` or `/Qopt-prefetch`	On IA-64 architecture, prefetch insertion optimization is enabled.
IA-32 architecture and Intel® 64 architecture: `-no-opt-prefetch` or `/Qopt-prefetch-`	On IA-32 architecture and Intel® 64 architecture, prefetch insertion optimization is disabled.

Description

This option enables or disables prefetch insertion optimization. The goal of prefetching is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache.

On IA-64 architecture, this option is enabled by default if you specify option `O1` or higher. To disable prefetching at these optimization levels, specify `-no-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch-` (Windows).

On IA-32 architecture and Intel® 64 architecture, this option enables prefetching when higher optimization levels are specified.

Alternate Options

Linux and Mac OS X: `-prefetch` (this is a deprecated option)

Windows: `/Qprefetch` (this is a deprecated option)

opt-prefetch-initial-values, Qopt-prefetch-initial-values

Enables or disables prefetches that are issued before a loop is entered.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-opt-prefetch-initial-values
-no-opt-prefetch-initial-values
```

Mac OS X:

```
None
```

Windows:

```
/Qopt-prefetch-initial-values
/Qopt-prefetch-initial-values-
```

Arguments

None

Default

`-opt-prefetch-initial-values` or `/Qopt-prefetch-initial-values` Prefetches are issued before a loop is entered.

Description

This option enables or disables prefetches that are issued before a loop is entered. These prefetches target the initial iterations of the loop.

When `-O1` or higher is specified on IA-64 architecture, prefetches are issued before a loop is entered. To disable these prefetches, specify `-no-opt-prefetch-initial-values` (Linux) or `/Qopt-prefetch-initial-values-` (Windows).

Alternate Options

None

opt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint

Determines whether the compiler issues prefetches for stores with exclusive hint.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-opt-prefetch-issue-excl-hint
-no-opt-prefetch-issue-excl-hint
```

Mac OS X:

None

Windows:

/Qopt-prefetch-issue-excl-hint

/Qopt-prefetch-issue-excl-hint-

Arguments

None

Default

-no-opt-prefetch-issue-excl-hint or/Qopt-prefetch-issue-excl-hint-	The compiler does not issue prefetches for stores with exclusive hint.

Description

This option determines whether the compiler issues prefetches for stores with exclusive hint. If option -opt-prefetch-issue-excl-hint (Linux) or /Qopt-prefetch-issue-excl-hint (Windows) is specified, the prefetches will be issued if the compiler determines it is beneficial to do so.

When prefetches are issued for stores with exclusive-hint, the cache-line is in "exclusive-mode". This saves on cache-coherence traffic when other processors try to access the same cache-line. This feature can improve performance tuning.

Alternate Options

None

opt-prefetch-next-iteration, Qopt-prefetch-next-iteration

Enables or disables prefetches for a memory access in the next iteration of a loop.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-prefetch-next-iteration`

`-no-opt-prefetch-next-iteration`

Mac OS X:

`None`

Windows:

`/Qopt-prefetch-next-iteration`

`/Qopt-prefetch-next-iteration-`

Arguments

None

Default

`-opt-prefetch-next-iteration` or `/Qopt-prefetch-next-iteration`	Prefetches are issued for a memory access in the next iteration of a loop.

Description

This option enables or disables prefetches for a memory access in the next iteration of a loop. It is typically used in a pointer-chasing loop.

When `-O1` or higher is specified on IA-64 architecture, prefetches are issued for a memory access in the next iteration of a loop. To disable these prefetches, specify `-no-opt-prefetch-next-iteration` (Linux) or `/Qopt-prefetch-next-iteration-` (Windows).

Alternate Options

None

opt-ra-region-strategy, Qopt-ra-region-strategy

Selects the method that the register allocator uses to partition each routine into regions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-ra-region-strategy[=keyword]`

Windows:

`/Qopt-ra-region-strategy[:keyword]`

Arguments

keyword	Is the method used for partitioning. Possible values are:	
	`routine`	Creates a single region for each routine.
	`block`	Partitions each routine into one region per basic block.
	`trace`	Partitions each routine into one region per trace.
	`region`	Partitions each routine into one region per loop.
	`default`	The compiler determines which method is used for partitioning.

Default

`-opt-ra-region-strategy=default` or `/Qopt-ra-region-strategy:default`	The compiler determines which method is used for partitioning. This is also the default if `keyword` is not specified.

Description

This option selects the method that the register allocator uses to partition each routine into regions.

When setting `default` is in effect, the compiler attempts to optimize the tradeoff between compile-time performance and generated code performance.

This option is only relevant when optimizations are enabled (`O1` or higher).

Alternate Options

None

See Also

-
-
- `O`

opt-report, Qopt-report

Tells the compiler to generate an optimization report to stderr.

IDE Equivalent

Windows: Diagnostics > Optimization Diagnostics Level

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report [n]`

Windows:

`/Qopt-report[:n]`

Arguments

n	Is the level of detail in the report. On Linux OS and Mac OS X systems, a space must appear before the *n*. Possible values are:
	`0` Tells the compiler to generate no optimization report.
	`1` Tells the compiler to generate a report with the minimum level of detail.
	`2` Tells the compiler to generate a report with the medium level of detail.
	`3` Tells the compiler to generate a report with the maximum level of detail.

Default

`-opt-report 2` or `/Qopt-report:2`	If you do not specify *n*, the compiler generates a report with medium detail. If you do not specify the option on the command line, the compiler does not generate an optimization report.

Description

This option tells the compiler to generate an optimization report to `stderr`.

Alternate Options

None

See Also

-
-
- `opt-report-file`, `Qopt-report-file`

Optimizing Applications: Optimizer Report Generation

opt-report-file, Qopt-report-file

Specifies the name for an optimization report.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-file=`*`file`*

Windows:

`/Qopt-report-file:`*`file`*

Arguments

`file`	Is the name for the optimization report.

Default

OFF	No optimization report is generated.

Description

This option specifies the name for an optimization report. If you use this option, you do not have to specify `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

Optimizing Applications: Optimizer Report Generation

opt-report-help, Qopt-report-help

Displays the optimizer phases available for report generation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-help`

Windows:

`/Qopt-report-help`

Arguments

None

Default

OFF	No optimization reports are generated.

Description

This option displays the optimizer phases available for report generation using `-opt-report-phase` (Linux and Mac OS X) or `/Qopt-report-phase` (Windows). No compilation is performed.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`
- `opt-report-phase`, `Qopt-report-phase`

opt-report-phase, Qopt-report-phase

Specifies an optimizer phase to use when optimization reports are generated.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-phase=`*`phase`*

Windows:

`/Qopt-report-phase:`*`phase`*

Arguments

`phase` Is the phase to generate reports for. Some of the possible values are:

`ipo`	The Interprocedural Optimizer phase
`hlo`	The High Level Optimizer phase
`hpo`	The High Performance Optimizer phase
`ilo`	The Intermediate Language Scalar Optimizer phase
`ecg`	The Code Generator phase (Windows and Linux systems using IA-64 architecture only)
`ecg_swp`	The software pipelining component of the Code Generator phase (Windows and Linux systems using IA-64 architecture only)
`pgo`	The Profile Guided Optimization phase
`all`	All optimizer phases

Default

OFF	No optimization reports are generated.

Description

This option specifies an optimizer phase to use when optimization reports are generated. To use this option, you must also specify `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

This option can be used multiple times on the same command line to generate reports for multiple optimizer phases.

When one of the logical names for optimizer phases is specified for phase, all reports from that optimizer phase are generated.

To find all phase possibilities, use option `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows).

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

opt-report-routine, Qopt-report-routine

Tells the compiler to generate reports on the routines containing specified text.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-routine=`*`string`*

Windows:

`/Qopt-report-routine:`*`string`*

Arguments

`string`	Is the text (string) to look for.

Default

OFF	No optimization reports are generated.

Description

This option tells the compiler to generate reports on the routines containing specified text as part of their name.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

opt-streaming-stores, Qopt-streaming-stores

Enables generation of streaming stores for optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-streaming-stores` *keyword*

Windows:

`/Qopt-streaming-stores:`*keyword*

Arguments

keyword	Specifies whether streaming stores are generated. Possible values are:	
	`always`	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound.
	`never`	Disables generation of streaming stores for optimization. Normal stores are performed.
	`auto`	Lets the compiler decide which instructions to use.

Default

`-opt-streaming-stores auto` or `/Qopt-streaming-stores:auto`	The compiler decides whether to use streaming stores or normal stores.

Description

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.

For this option to be effective, the compiler must be able to generate SSE2 (or higher) instructions. For more information, see compiler option `x` or `ax`.

This option may be useful for applications that can benefit from streaming stores.

Alternate Options

None

See Also

-
-
- ax, Qax
- x, Qx
- opt-mem-bandwidth, Qopt-mem-bandwidth, Qx

Optimizing Applications: Vectorization Support

opt-subscript-in-range, Qopt-subscript-in-range

Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-opt-subscript-in-range
```

```
-no-opt-subscript-in-range
```

Windows:

```
/Qopt-subscript-in-range
```

```
/Qopt-subscript-in-range-
```

Arguments

None

Default

`-no-opt-subscript-in-range` or `/Qopt-subscript-in-range-`	The compiler assumes overflows in the intermediate computation of subscript expressions in loops.

Description

This option determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.

If you specify `-opt-subscript-in-range` (Linux and Mac OS X) or `/Qopt-subscript-in-range` (Windows), the compiler ignores any data type conversions used and it assumes no overflows in the intermediate computation of subscript expressions. This feature can enable more loop transformations.

Alternate Options

None

Example

The following shows an example where these options can be useful. m is declared as type integer(kind=8) (64-bits) and all other variables inside the subscript are declared as type integer(kind=4) (32-bits):

```
A[ i  + j  + ( n  + k) * m ]
```

optimize

See O.

Os

Enables optimizations that do not increase code size and produces smaller code size than O2.

IDE Equivalent

Windows: **Optimization > Favor Size or Speed**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Os`

Windows:

`/Os`

Arguments

None

Default

OFF	Optimizations are made for code speed. However, if `O1` is specified, `Os` is the default.

Description

This option enables optimizations that do not increase code size and produces smaller code size than `O2`. It disables some optimizations that increase code size for a small speed benefit.

This option tells the compiler to favor transformations that reduce code size over transformations that produce maximum performance.

Alternate Options

None

See Also

-
- `O`
- `Ot`

Ot

Enables all speed optimizations.

IDE Equivalent

Windows: **Optimization > Favor Size or Speed (`/Ot, /Os`)**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Ot`

Arguments

None

Default

`/Ot`	Optimizations are made for code speed. If `Od` is specified, all optimizations are disabled. If `O1` is specified, `Os` is the default.

Description

This option enables all speed optimizations.

Alternate Options

None

See Also

•

- O
- Os

Ox

See O.

fomit-frame-pointer, Oy

Determines whether EBP is used as a general-purpose register in optimizations.

IDE Equivalent

Windows: **Optimization > Omit Frame Pointers**

Linux: None

Mac OS X: **Optimization > Provide Frame Pointer**

Architectures

`-f[no-]omit-frame-pointer`: IA-32 architecture, Intel® 64 architecture

`/Oy[-]`: IA-32 architecture

Syntax

Linux and Mac OS X:

```
-fomit-frame-pointer
```

```
-fno-omit-frame-pointer
```

Windows:

```
/Oy
```

```
/Oy-
```

Arguments

None

Default

`-fomit-frame-pointer` or `/Oy`	EBP is used as a general-purpose register in optimizations. However, on Linux* and Mac OS X systems, the default is `-fno-omit-frame-pointer` if option `-O0` or `-g` is specified. On Windows* systems, the default is `/Oy-` if option `/Od` is specified.

Description

These options determine whether EBP is used as a general-purpose register in optimizations. Options `-fomit-frame-pointer` and `/Oy` allow this use. Options `-fno-omit-frame-pointer` and `/Oy-` disallow it.

Some debuggers expect EBP to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-fno-omit-frame-pointer` and `/Oy-` options direct the compiler to generate code that maintains and uses EBP as a stack frame pointer for all functions so that a debugger can still produce a stack backtrace without doing the following:

- For `-fno-omit-frame-pointer`: turning off optimizations with `-O0`
- For `/Oy-`: turning off `/O1`, `/O2`, or `/O3` optimizations

The `-fno-omit-frame-pointer` option is set when you specify option `-O0` or the `-g` option. The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3`.

The `/Oy` option is set when you specify the `/O1`, `/O2`, or `/O3` option. Option `/Oy-` is set when you specify the `/Od` option.

Using the `-fno-omit-frame-pointer` or `/Oy-` option reduces the number of available general-purpose registers by 1, and can result in slightly less efficient code.

Alternate Options

Linux and Mac OS X: `-fp` (this is a deprecated option)

Windows: None

p

Compiles and links for function profiling with gprof(1).

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-p`

Windows:

`None`

Arguments

None

Default

OFF	Files are compiled and linked without profiling.

Description

This option compiles and links for function profiling with gprof(1).

Alternate Options

Linux and Mac OS X: `-pg` (only available on systems using IA-32 architecture or Intel® 64 architecture), `-qp` (this is a deprecated option)

Windows: None

P

See preprocess-only.

pad, Qpad

Enables the changing of the variable and array memory layout.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-pad`

`-nopad`

Windows:

`/Qpad`

`/Qpad-`

Arguments

None

Default

`-nopad` or `/Qpad-` Variable and array memory layout is performed by default methods.

Description

This option enables the changing of the variable and array memory layout.

This option is effectively not different from the `align` option when applied to structures and derived types. However, the scope of `pad` is greater because it applies also to common blocks, derived types, sequence types, and structures.

Alternate Options

None

See Also

-
-
- `align`

pad-source, Qpad-source

Specifies padding for fixed-form source records.

IDE Equivalent

Windows: **Language > Pad Fixed Form Source Lines**

Linux: None

Mac OS X: **Language > Pad Fixed Form Source Lines**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-pad-source
-nopad-source
```

Windows:

```
/pad-source
/nopad-source
/Qpad-source
/Qpad-source-
```

Arguments

None

Default

`-nopad-source` or `/Qpad-` Fixed-form source records are not padded.
`source-`

Description

This option specifies padding for fixed-form source records. It tells the compiler that fixed-form source lines shorter than the statement field width are to be padded with spaces to the end of the statement field. This affects the interpretation of character and Hollerith literals that are continued across source records.

The default value setting causes a warning message to be displayed if a character or Hollerith literal that ends before the statement field ends is continued onto the next source record. To suppress this warning message, specify option `-warn nousage` (Linux and Mac OS X) or `/warn:nousage` (Windows).

Specifying `pad-source` or `/Qpad-source` can prevent warning messages associated with option `-warn usage` (Linux and Mac OS X) or `/warn:usage` (Windows).

Alternate Options

None

See Also

-
-
- `warn`

par-affinity, Qpar-affinity

Specifies thread affinity.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-par-affinity=[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

Mac OS X:

`None`

Windows:

`/Qpar-affinity:[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

Arguments

`modifier`	Is one of the following values: `granularity={fine|thread|core}`, `[no]respect`, `[no]verbose`, `[no]warnings`, `proclist=proc_list`. The default is `granularity=core, respect, and noverbose`. For information on value `proclist`, see Thread Affinity Interface in *Optimizing Applications*.
`type`	Indicates the thread affinity. This argument is required and must be one of the following values: `compact`, `disabled`, `explicit`, `none`, `scatter`, `logical`, `physical`. The default is `none`. Values `logical` and `physical` are deprecated. Use `compact` and `scatter`, respectively, with no *`permute`* value.
`permute`	Is a positive integer. You cannot use this argument with *`type`* setting `explicit`, `none`, or `disabled`. The default is 0.
`offset`	Is a positive integer. You cannot use this argument with *`type`* setting `explicit`, `none`, or `disabled`. The default is 0.

Default

OFF	The thread affinity is determined by the run-time environment.

Description

This option specifies thread affinity, which binds threads to physical processing units. It has the same effect as environment variable KMP_AFFINITY.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- Linux* OS: You have specified option `-parallel` or `-openmp` (or both).

  Windows* OS: You have specified option `/Qparallel` or `/Qopenmp` (or both).
- You are compiling the main program.

Alternate Options

None

See Also

-
-
- Thread Affinity Interface

par-num-threads, Qpar-num-threads

Specifies the number of threads to use in a parallel region.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-num-threads=`*n*

Windows:

`/Qpar-num-threads:`*n*

Arguments

n	Is the number of threads to use. It must be a positive integer.

Default

OFF	The number of threads to use is determined by the run-time environment.

Description

This option specifies the number of threads to use in a parallel region. It has the same effect as environment variable OMP_NUM_THREADS.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- Linux* OS and Mac OS* X: You have specified option `-parallel` or `-openmp` (or both).

  Windows* OS: You have specified option `/Qparallel` or `/Qopenmp` (or both).
- You are compiling the main program.

Alternate Options

None

par-report, Qpar-report

Controls the diagnostic information reported by the auto-parallelizer.

IDE Equivalent

Windows: **Compilation Diagnostics > Auto-Parallelizer Diagnostic Level**

Linux: None

Mac OS X: **Diagnostics > Auto-Parallelizer Report**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-report[`*n*`]`

Windows:

`/Qpar-report[n]`

Arguments

n	Is a value denoting which diagnostic messages to report. Possible values are:
0	Tells the auto-parallelizer to report no diagnostic information.
1	Tells the auto-parallelizer to report diagnostic messages for loops successfully auto-parallelized. The compiler also issues a "LOOP AUTO-PARALLELIZED" message for parallel loops.
2	Tells the auto-parallelizer to report diagnostic messages for loops successfully and unsuccessfully auto-parallelized.
3	Tells the auto-parallelizer to report the same diagnostic messages specified by 2 plus additional information about any proven or assumed dependencies inhibiting auto-parallelization (reasons for not parallelizing).

Default

`-par-report1` or `/Qpar-report1`	If you do not specify *n*, the compiler displays diagnostic messages for loops successfully auto-parallelized. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the diagnostic information reported by the auto-parallelizer (parallel optimizer). To use this option, you must also specify `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows).

If this option is specified on the command line, the report is sent to `stdout`.

On Windows systems, if this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

par-runtime-control, Qpar-runtime-control

Generates code to perform run-time checks for loops that have symbolic loop bounds.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-runtime-control`

`-no-par-runtime-control`

Windows:

`/Qpar-runtime-control`

`/Qpar-runtime-control-`

Arguments

None

Default

`-no-par-runtime-control` The compiler uses default heuristics when checking loops.
or `/Qpar-runtime-control-`

Description

This option generates code to perform run-time checks for loops that have symbolic loop bounds.

If the granularity of a loop is greater than the parallelization threshold, the loop will be executed in parallel.

If you do not specify this option, the compiler may not parallelize loops with symbolic loop bounds if the compile-time granularity estimation of a loop can not ensure it is beneficial to parallelize the loop.

Alternate Options

None

par-schedule, Qpar-schedule

Lets you specify a scheduling algorithm or a tuning method for loop iterations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-schedule-`*`keyword`*`[=`*`n`*`]`

Windows:

`/Qpar-schedule-`*`keyword`*`[[:]`*`n`*`]`

Arguments

`keyword`	Specifies the scheduling algorithm or tuning method. Possible values are:	
	`auto`	Lets the compiler or run-time system determine the scheduling algorithm.
	`static`	Divides iterations into contiguous pieces.
	`static-balanced`	Divides iterations into even-sized chunks.
	`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
	`dynamic`	Gets a set of iterations dynamically.

`guided`	Specifies a minimum number of iterations.
`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
`runtime`	Defers the scheduling decision until run time.

n Is the size of the chunk or the number of iterations for each chunk. This setting can only be specified for static, dynamic, and guided. For more information, see the descriptions of each keyword below.

Default

static-balanced Iterations are divided into even-sized chunks and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

Description

This option lets you specify a scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.

This option affects performance tuning and can provide better performance during auto-parallelization.

Option	Description
`-par-schedule-auto` or `/Qpar-schedule-auto`	Lets the compiler or run-time system determine the scheduling algorithm. Any possible mapping may occur for iterations to threads in the team.
`-par-schedule-static` or `/Qpar-schedule-static`	Divides iterations into contiguous pieces (chunks) of size *n*. The chunks are assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations.

Option	Description
	If no *n* is specified, the iteration space is divided into chunks that are approximately equal in size, and each thread is assigned at most one chunk.
`-par-schedule-static-balanced` or `/Qpar-schedule-static-balanced`	Divides iterations into even-sized chunks. The chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
`-par-schedule-static-steal` or `/Qpar-schedule-static-steal`	Divides iterations into even-sized chunks, but when a thread completes its chunk, it can steal parts of chunks assigned to neighboring threads. Each thread keeps track of L and U, which represent the lower and upper bounds of its chunks respectively. Iterations are executed starting from the lower bound, and simultaneously, L is updated to represent the new lower bound.
`-par-schedule-dynamic` or `/Qpar-schedule-dynamic`	Can be used to get a set of iterations dynamically. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations. Each chunk contains *n* iterations, except for the last chunk to be assigned, which may have fewer iterations. If no *n* is specified, the default is 1.
`-par-schedule-guided` or `/Qpar-schedule-guided`	Can be used to specify a minimum number of iterations. Assigns iterations to threads in chunks as the threads request them. The

Option	Description
	thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For an *n* with value *k* (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). If no *n* is specified, the default is 1.
`-par-schedule-guided-analytical` or `/Qpar-schedule-guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution. The method depends on run-time implementation. Loop bounds are calculated with faster synchronization and chunks are dynamically dispatched at run time by threads in the team.
`-par-schedule-runtime` or `/Qpar-schedule-runtime`	Defers the scheduling decision until run time. The scheduling algorithm and chunk size are then taken from the setting of environment variable OMP_SCHEDULE.

Alternate Options

None

par-threshold, Qpar-threshold

Sets a threshold for the auto-parallelization of loops.

IDE Equivalent

Windows: **Optimization > Threshold For Auto-Parallelization**

Linux: None

Mac OS X: **Optimization > Threshold For Auto-Parallelization**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-threshold[n]`

Windows:

`/Qpar-threshold[[:]n]`

Arguments

n	Is an integer whose value is the threshold for the auto-parallelization of loops. Possible values are 0 through 100. If *n* is 0, loops get auto-parallelized always, regardless of computation work volume. If *n* is 100, loops get auto-parallelized when performance gains are predicted based on the compiler analysis data. Loops get auto-parallelized only if profitable parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to parallelize only if there is a 50% probability of the code speeding up if executed in parallel.

Default

`-par-threshold100` or `/Qpar-threshold100`	Loops get auto-parallelized only if profitable parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. To use this option, you must also specify `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows).

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Alternate Options

None

parallel, Qparallel

Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

IDE Equivalent

Windows: **Optimization > Parallelization**

Linux: None

Mac OS X: **Optimization > Parallelization**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-parallel`

Windows:

`/Qparallel`

Arguments

None

Default

OFF	Multithreaded code is not generated for loops that can be safely executed in parallel.

Description

This option tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

To use this option, you must also specify option `O2` or `O3`.

NOTE. On Mac OS X systems, when you enable automatic parallelization, you must also set the `DYLD_LIBRARY_PATH` environment variable within Xcode or an error will be displayed.

Alternate Options

None

See Also

-
-
- `par-report, Qpar-report`
- `par-affinity, Qpar-affinity`
- `par-num-threads, Qpar-num-threads`
- `par-runtime-control, Qpar-runtime-control`
- `par-schedule, Qpar-schedule`
- `O`

pc, Qpc

Enables control of floating-point significand precision.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-pc`*n*

Windows:

`/Qpc`*n*

Arguments

n	Is the floating-point significand precision. Possible values are:
`32`	Rounds the significand to 24 bits (single precision).
`64`	Rounds the significand to 53 bits (double precision).
`80`	Rounds the significand to 64 bits (extended precision).

Default

`-pc80` or `/Qpc64`	On Linux* and Mac OS* X systems, the floating-point significand is rounded to 64 bits. On Windows* systems, the floating-point significand is rounded to 53 bits.

Description

This option enables control of floating-point significand precision.

Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the this option.

Note that a change of the default precision control or rounding mode, for example, by using the `-pc32` (Linux and Mac OS X) or `/Qpc32` (Windows) option or by user intervention, may affect the results returned by some of the mathematical functions.

Alternate Options

None

See Also

-
-

Floating-point Operations: Floating-point Options Quick Reference

pdbfile

Specifies that any debug information generated by the compiler should be saved to a program database file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/pdbfile[:`*`file`*`]`

`/nopdbfile`

Arguments

file	Is the name of the program database file.

Default

/nopdbfile	Debug information generated by the compiler is not saved to a program database file.

Description

This option specifies that any debug information generated by the compiler should be saved to a program database file. To use this option, you must also specify /debug:full (or the equivalent).

If *file* is not specified, the default file name used is the name of your file with an extension of .pdb.

The compiler places debug information in the object file if you specify /nopdbfile or omit both /pdbfile and /debug:full (or the equivalent).

Alternate Options

None

See Also

-
- debug (Windows*)

pg

See p.

pie

Produces a position-independent executable on processors that support it.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-pie`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	The driver does not set up special run-time libraries and the linker does not perform the optimizations on executables.

Description

This option produces a position-independent executable on processors that support it. It is both a compiler option and a linker option. When used as a compiler option, this option ensures the linker sets up run-time libraries correctly.

Normally the object linked has been compiled with option `-fpie`.

When you specify `-pie`, it is recommended that you specify the same options that were used during compilation of the object.

Alternate Options

None

See Also

-
- fpie

prec-div, Qprec-div

Improves precision of floating-point divides.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prec-div`

`-no-prec-div`

Windows:

`/Qprec-div`

`/Qprec-div-`

Arguments

None

Default

`-prec-div` or `/Qprec-div`	The compiler uses this method for floating-point divides.

Description

This option improves precision of floating-point divides. It has a slight impact on speed.

With some optimizations, such as `-xSSE2` (Linux) or `/QxSSE2` (Windows), the compiler may change floating-point division computations into multiplication by the reciprocal of the denominator. For example, A/B is computed as A * (1/B) to improve the speed of the computation.

However, sometimes the value produced by this transformation is not as accurate as full IEEE division. When it is important to have fully precise IEEE division, use this option to disable the floating-point division-to-multiplication optimization. The result is more accurate, with some loss of performance.

If you specify `-no-prec-div` (Linux and Mac OS X) or `/Qprec-div-` (Windows), it enables optimizations that give slightly less precise results than full IEEE division.

Alternate Options

None

See Also

-
-

Floating-point Operations: Floating-point Options Quick Reference

prec-sqrt, Qprec-sqrt

Improves precision of square root implementations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-prec-sqrt
```

```
-no-prec-sqrt
```

Windows:

```
/Qprec-sqrt
```

```
/Qprec-sqrt-
```

Arguments

None

Default

`-no-prec-sqrt` or `/Qprec-sqrt-`	The compiler uses a faster but less precise implementation of square root. However, the default is `-prec-sqrt` or `/Qprec-sqrt` if any of the following options are specified: `/Od`, `/Op`, or `/Qprec` on Windows systems; `-O0`, `-mp` (or `-fltconsistency`), or `-mp1` on Linux and Mac OS X systems.

Description

This option improves precision of square root implementations. It has a slight impact on speed.

This option inhibits any optimizations that can adversely affect the precision of a square root computation. The result is fully precise square root implementations, with some loss of performance.

Alternate Options

None

preprocess-only

Causes the Fortran preprocessor to send output to a file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-preprocess-only
```

Windows:

```
/preprocess-only
```

Arguments

None

Default

OFF	Preprocessed source files are output to the compiler.

Description

This option causes the Fortran preprocessor to send output to a file.

The source file is preprocessed by the Fortran preprocessor, and the result for each source file is output to a corresponding .i or .i90 file.

Note that the source file is not compiled.

Alternate Options

Linux and Mac OS X: `-P`

Windows: `/P`

print-multi-lib

Prints information about where system libraries should be found.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-print-multi-lib
```

Windows:

```
None
```

Arguments

None

Default

OFF	No information is printed unless the option is specified.

Description

This option prints information about where system libraries should be found, but no compilation occurs. It is provided for compatibility with gcc.

Alternate Options

None

prof-data-order, Qprof-data-order

Enables or disables data ordering if profiling information is enabled.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

```
-prof-data-order
-no-prof-data-order
```

Mac OS X:

```
None
```

Windows:

```
/Qprof-data-order
/Qprof-data-order-
```

Arguments

None

Default

`-no-prof-data-order` or `/Qprof-data-order-`	Data ordering is disabled.

Description

This option enables or disables data ordering if profiling information is enabled. It controls the use of profiling information to order static program data items.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows).
- For feedback compilation, you must specify `-prof-use` (Linux) or `/Qprof-use` (Windows). You must not use multi-file optimization by specifying options such as option `-ipo` (Linux) or `/Qipo` (Windows), or option `-ipo-c` (Linux) or `/Qipo-c` (Windows).

Alternate Options

None

See Also

-
-
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-func-order, Qprof-func-order

prof-dir, Qprof-dir

Specifies a directory for profiling information output files.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-dir dir`

Windows:

`/Qprof-dir dir`

Arguments

dir	Is the name of the directory.

Default

OFF	Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Alternate Options

None

See Also

-
-

Floating-point Operations:

Profile-guided Optimization (PGO) Quick Reference

Coding Guidelines for Intel(R) Architectures

prof-file, Qprof-file

Specifies an alternate file name for the profiling summary files.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-file` *file*

Windows:

`/Qprof-file` *file*

Arguments

file	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*

Description

This option specifies an alternate file name for the profiling summary files. The *file* is used as the base name for files created by different profiling passes.

If you add this option to profmerge, the .dpi file will be named *file*.dpi instead of pgopti.dpi.

If you specify `-prof-genx` (Linux and Mac OS X) or `/Qprof-genx` (Windows) with this option, the .spi and .spl files will be named *file*.spi and *file*.spl instead of pgopti.spi and pgopti.spl.

If you specify `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) with this option, the .dpi file will be named *file*.dpi instead of pgopti.dpi.

Alternate Options

None

See Also

-
-
- prof-gen, Qprof-gen
- prof-use, Qprof-use

Optimizing Applications:

Profile-guided Optimizations Overview

Coding Guidelines for Intel(R) Architectures

Profile an Application

prof-func-groups

Enables or disables function grouping if profiling information is enabled.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux:

```
-prof-func-groups
```

```
-no-prof-func-groups
```

Mac OS X:

```
None
```

Windows:

```
None
```

Arguments

None

Default

`-no-prof-func-groups` Function grouping is disabled.

Description

This option enables or disables function grouping if profiling information is enabled.

A "function grouping" is a profiling optimization in which entire routines are placed either in the cold code section or the hot code section.

If profiling information is enabled by option `-prof-use`, option `-prof-func-groups` is set and function grouping is enabled. However, if you explicitly enable `-prof-func-order` (Linux) or `/Qprof-func-order` (Windows), function ordering is performed instead of function grouping.

If you want to disable function grouping when profiling information is enabled, specify `-no-prof-func-groups`.

To set the hotness threshold for function grouping, use option `-prof-hotness-threshold` (Linux) or `/Qprof-hotness-threshold` (Windows).

Alternate Options

`-func-groups` (this is a deprecated option)

See Also

-
- `prof-use, Qprof-use`
- `prof-func-order, Qprof-func-order`
- `prof-hotness-threshold, Qprof-hotness-threshold`

prof-func-order, Qprof-func-order

Enables or disables function ordering if profiling information is enabled.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-prof-func-order`

`-no-prof-func-order`

Mac OS X:

`None`

Windows:

`/Qprof-func-order`

`/Qprof-func-order-`

Arguments

None

Default

`-no-prof-func-order` or `/Qprof-func-order-`	Function ordering is disabled.

Description

This option enables or disables function ordering if profiling information is enabled.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows).
- For feedback compilation, you must specify `-prof-use` (Linux) or `/Qprof-use` (Windows). You must not use multi-file optimization by specifying options such as option `-ipo` (Linux) or `/Qipo` (Windows), or option `-ipo-c` (Linux) or `/Qipo-c` (Windows).

If you enable profiling information by specifying option `-prof-use` (Linux) or `/Qprof-use` (Windows), `-prof-func-groups` (Linux) and `/Qprof-func-groups` (Windows) are set and function grouping is enabled. However, if you explicitly enable `-prof-func-order` (Linux) or `/Qprof-func-order` (Windows), function ordering is performed instead of function grouping.

On Linux* systems, this option is only available for Linux linker 2.15.94.0.1, or later.

To set the hotness threshold for function grouping and function ordering, use option `-prof-hotness-threshold` (Linux) or `/Qprof-hotness-threshold` (Windows).

Alternate Options

None

The following example shows how to use this option on a Windows system:

```
ifort /Qprof-gen:globdata file1.f90 file2.f90 /exe:instrumented.exe
    ./instrumented.exe
ifort /Qprof-use /Qprof-func-order file1.f90 file2.f90 /exe:feedback.exe
```

The following example shows how to use this option on a Linux system:

```
ifort -prof-gen:globdata file1.f90 file2.f90 -o instrumented
    ./instrumented.exe
ifort -prof-use -prof-func-order file1.f90 file2.f90 -o feedback
```

See Also

-
-
- `prof-hotness-threshold, Qprof-hotness-threshold`
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-data-order, Qprof-data-order
- prof-func-groups

prof-gen, Qprof-gen

Produces an instrumented object file that can be used in profile-guided optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-gen[=keyword]`

`-no-prof-gen`

Windows:

`/Qprof-gen[:keyword]`

`/Qprof-gen-`

Arguments

keyword	Specifies details for the instrumented file. Possible values are:	
	`default`	Produces an instrumented object file. This is the same as specifying `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) with no keyword.
	`srcpos`	Produces an instrumented object file that includes extra source position information. This option is the same as option `-prof-genx` (Linux* and Mac OS* X) or `/Qprof-genx` (Windows*), which are deprecated.
	`globdata`	Produces an instrumented object file that includes information for global data layout.

Default

`-no-prof-gen` or `/Qprof-gen-` Profile generation is disabled.

Description

This option produces an instrumented object file that can be used in profile-guided optimization. It gets the execution count of each basic block.

If you specify keyword `srcpos` or `globdata`, a static profile information file (.spi) is created. These settings may increase the time needed to do a parallel build using -prof-gen, because of contention writing the .spi file.

These options are used in phase 1 of the Profile Guided Optimizer (PGO) to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution.

Alternate Options

None

See Also

•

•

Optimizing Applications:

Basic PGO Options

Example of Profile-Guided Optimization

prof-genx, Qprof-genx

This is a deprecated option. See prof-gen keyword `srcpos`*.*

prof-hotness-threshold, Qprof-hotness-threshold

Lets you set the hotness threshold for function grouping and function ordering.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-prof-hotness-threshold=`*n*

Mac OS X:

`None`

Windows:

`/Qprof-hotness-threshold:n`

Arguments

n	Is the hotness threshold. *n* is a percentage having a value between 0 and 100 inclusive. If you specify 0, there will be no hotness threshold setting in effect for function grouping and function ordering.

Default

OFF	The compiler's default hotness threshold setting of 10 percent is in effect for function grouping and function ordering.

Description

This option lets you set the hotness threshold for function grouping and function ordering.

The "hotness threshold" is the percentage of functions in the application that should be placed in the application's hot region. The hot region is the most frequently executed part of the application. By grouping these functions together into one hot region, they have a greater probability of remaining resident in the instruction cache. This can enhance the application's performance.

For this option to take effect, you must specify option `-prof-use` (Linux) or `/Qprof-use` (Windows) and one of the following:

- On Linux systems: `-prof-func-groups` or `-prof-func-order`
- On Windows systems: `/Qprof-func-order`

Alternate Options

None

See Also

-
-
- prof-use, Qprof-use
- prof-func-groups
- prof-func-order, Qprof-func-order

prof-src-dir, Qprof-src-dir

Determines whether directory information of the source file under compilation is considered when looking up profile data records.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-src-dir`

`-no-prof-src-dir`

Windows:

`/Qprof-src-dir`

`/Qprof-src-dir-`

Arguments

None

Default

`-prof-src-dir` or `/Qprof-src-dir`	Directory information is used when looking up profile data records in the .dpi file.

Description

This option determines whether directory information of the source file under compilation is considered when looking up profile data records in the .dpi file. To use this option, you must also specify option `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

If the option is enabled, directory information is considered when looking up the profile data records within the .dpi file. You can specify directory information by using one of the following options:

- Linux and Mac OS X: `-prof-src-root` or `-prof-src-root-cwd`
- Windows: `/Qprof-src-root` or `/Qprof-src-root-cwd`

If the option is disabled, directory information is ignored and only the name of the file is used to find the profile data record.

Note that options `-prof-src-dir` (Linux and Mac OS X) and `/Qprof-src-dir` (Windows) control how the names of the user's source files get represented within the .dyn or .dpi files. Options `-prof-dir` (Linux and Mac OS X) and `/Qprof-dir` (Windows) specify the location of the .dyn or the .dpi files.

Alternate Options

None

See Also

-
-
- `prof-use, Qprof-use`
- `prof-src-root, Qprof-src-root`
- `prof-src-root-cwd, Qprof-src-root-cwd`

prof-src-root, Qprof-src-root

Lets you use relative directory paths when looking up profile data and specifies a directory as the base.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-src-root=`*`dir`*

Windows:

`/Qprof-src-root:dir`

Arguments

dir	Is the base for the relative paths.

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It lets you specify a directory as the base. The paths are relative to a base directory specified during the `-prof-gen` (Linux and Mac OS X) or `/Qprof-gen` (Windows) compilation phase.

This option is available during the following phases of compilation:

- Linux and Mac OS X: `-prof-gen` and `-prof-use` phases
- Windows: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `-prof-gen` or `/Qprof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `-prof-use` or `/Qprof-use` phase, it specifies a root directory that replaces the root directory specified at the `-prof-gen` or `/Qprof-gen` phase for forming the lookup keys.

To be effective, this option or option `-prof-src-root-cwd` (Linux and Mac OS X) or `/Qprof-src-root-cwd` (Windows) must be specified during the `-prof-gen` or `/Qprof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

Alternate Options

None

Consider the initial `-prof-gen` compilation of the source file c:\user1\feature_foo\myproject\common\glob.f90:

```
ifort -prof-gen -prof-src-root=c:\user1\feature_foo\myproject -c common\glob.f90
```

For the -prof-use phase, the file glob.f90 could be moved into the directory c:\user2\feature_bar\myproject\common\glob.f90 and profile information would be found from the .dpi when using the following:

```
ifort -prof-use -prof-src-root=c:\user2\feature_bar\myproject -c common\glob.f90
```

If you do not use option -prof-src-root during the -prof-gen phase, by default, the -prof-use compilation can only find the profile data if the file is compiled in the c:\user1\feature_foo\my_project\common directory.

See Also

-
-
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-src-dir, Qprof-src-dir
- prof-src-root-cwd, Qprof-src-root-cwd

prof-src-root-cwd, Qprof-src-root-cwd

Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-prof-src-root-cwd
```

Windows:

```
/Qprof-src-root-cwd
```

Arguments

None

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It specifies the current working directory as the base. To use this option, you must also specify option `-prof-use` (Linux and Mac OS) or `/Qprof-use` (Windows).

This option is available during the following phases of compilation:

- Linux and Mac OS X: `-prof-gen` and `-prof-use` phases
- Windows: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `-prof-gen` or `/Qprof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `-prof-use` or `/Qprof-use` phase, it specifies a root directory that replaces the root directory specified at the `-prof-gen` or `/Qprof-gen` phase for forming the lookup keys.

To be effective, this option or option `-prof-src-root` (Linux and Mac OS X) or `/Qprof-src-root` (Windows) must be specified during the `-prof-gen` or `/Qprof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

Alternate Options

None

See Also

-
-
- `prof-gen, Qprof-gen`
- `prof-use, Qprof-use`
- `prof-src-dir, Qprof-src-dir`
- `prof-src-root, Qprof-src-root`

prof-use, Qprof-use

Enables the use of profiling information during optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-use[=arg]`

`-no-prof-use`

Windows:

`/Qprof-use[:arg]`

`/Qprof-use-`

Arguments

arg	Specifies additional instructions. Possible values are:	
	`weighted`	Tells the profmerge utility to apply a weighting to the .dyn file values when creating the .dpi file to normalize the data counts when the training runs have differentexecution durations. This argument only has an effect when the compiler invokes the profmerge utility to create the .dpi file. This argument does not have an effect if the .dpi file was previously created without weighting.
	`[no]merge`	Enables or disables automatic invocation of the profmerge utility. The default is `merge`. Note that you cannot specify both

	`weighted` and `nomerge`. If you try to specify both values, a warning will be displayed and `nomerge` takes precedence.
`default`	Enables the use of profiling information during optimization. The profmerge utility is invoked by default. This value is the same as specifying `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) with no argument.

Default

`-no-prof-use` or `/Qprof-` Profiling information is not used during optimization.
`use-`

Description

This option enables the use of profiling information (including function splitting and function grouping) during optimization. It enables option `-fnsplit` (Linux) or `/Qfnsplit` (Windows).

This option instructs the compiler to produce a profile-optimized executable and it merges available profiling output files into a pgopti.dpi file.

Note that there is no way to turn off function grouping if you enable it using this option.

To set the hotness threshold for function grouping and function ordering, use option `-prof-hotness-threshold` (Linux) or `/Qprof-hotness-threshold` (Windows).

Alternate Options

None

See Also

-
-
- `prof-hotness-threshold, Qprof-hotness-threshold`

Optimizing Applications:

Basic PGO Options

Example of Profile-Guided Optimization

ansi-alias, Qansi-alias

Tells the compiler to assume that the program adheres to Fortran Standard type aliasability rules.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ansi-alias`

`-no-ansi-alias`

Windows:

`Qansi-alias`

`Qansi-alias-`

Arguments

None

Default

`-ansi-alias` or `/Qansi-alias`	Programs adhere to Fortran Standard type aliasability rules.

Description

This option tells the compiler to assume that the program adheres to type aliasability rules defined in the Fortran Standard.

For example, an object of type real cannot be accessed as an integer. For information on the rules for data types and data type constants, see "Data Types, Constants, and Variables" in the Language Reference.

This option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.
- References to objects of two different scalar types cannot alias. For example, an object of type integer cannot alias with an object of type real or an object of type real cannot alias with an object of type double precision.

If your program adheres to the Fortran Standard type aliasability rules, this option enables the compiler to optimize more aggressively. If it doesn't adhere to these rules, then you should disable the option with `-no-ansi-alias` (Linux and Mac OS X) or `/Qansi-alias-` (Windows) so the compiler does not generate incorrect code.

Alternate Options

None

auto, Qauto

See automatic.

auto-scalar, Qauto-scalar

Causes scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute to be allocated to the run-time stack.

IDE Equivalent

Windows: **Data > Local Variable Storage** (`/Qsave`, `/Qauto`, `/Qauto_scalar`)

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-auto-scalar
```

Windows:

`/Qauto-scalar`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL that do not have the SAVE attribute are allocated to the run-time stack. Note that if option `recursive`, `-openmp` (Linux and Mac OS X), or `/Qopenmp` (Windows) is specified, the default is `automatic`.

Description

This option causes allocation of scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL to the run-time stack. It is as if they were declared with the AUTOMATIC attribute.

It does not affect variables that have the SAVE attribute (which include initialized locals) or that appear in an EQUIVALENCE statement or in a common block.

This option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a SAVE statement.

You cannot specify option `save`, `auto`, or `automatic` with this option.

NOTE. On Windows NT* systems, there is a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/automatic`, `/auto`, or `/Qauto` because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty. `/Qauto-scalar` enables the compiler to make better choices about which variables should be kept in registers during program execution.

Alternate Options

None

See Also

-
-
- auto
- save

autodouble, Qautodouble

See real-size.

ax, Qax

Tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit.

IDE Equivalent

Windows: **Code Generation > Add Processor-Optimized Code Path**

Optimization > Generate Alternate Code Paths

Linux: None

Mac OS X: **Code Generation > Add Processor-Optimized Code Path**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-ax`*`processor`*

Windows:

`/Qax`*`processor`*

Arguments

processor	Indicates the processor for which code is generated. The following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`SSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
`SSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.
`SSSE3`	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. For Mac OS* X systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
`SSE3`	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. For Mac OS* X systems, this value is only supported on IA-32 architecture. This replaces value P, which is deprecated.
`SSE2`	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel®

	SSE2. This value is not available on Mac OS* X systems. This replaces value N, which is deprecated.

Default

OFF	No auto-dispatch code is generated. Processor-specific code is generated and is controlled by the setting of compiler option `-m` (Linux), compiler option `/arch` (Windows), or compiler option `-x` (Mac OS* X).

Description

This option tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit. It also generates a baseline code path. The baseline code is usually slower than the specialized code.

The baseline code path is determined by the architecture specified by the `-x` (Linux and Mac OS X) or `/Qx` (Windows) option. While there are defaults for the `-x` or `/Qx` option that depend on the operating system being used, you can specify an architecture for the baseline code that is higher or lower than the default. The specified architecture becomes the effective minimum architecture for the baseline code path.

If you specify both the `-ax` and `-x` options (Linux and Mac OS X) or the `/Qax` and `/Qx` options (Windows), the baseline code will only execute on processors compatible with the processor type specified by the `-x` or `/Qx` option.

This option tells the compiler to find opportunities to generate separate versions of functions that take advantage of features of the specified Intel® processor.

If the compiler finds such an opportunity, it first checks whether generating a processor-specific version of a function is likely to result in a performance gain. If this is the case, the compiler generates both a processor-specific version of a function and a baseline version of the function. At run time, one of the versions is chosen to execute, depending on the Intel processor in use. In this way, the program can benefit from performance gains on more advanced Intel processors, while still working properly on older processors.

You can use more than one of the processor values by combining them. For example, you can specify `-axSSE4.1,SSSE3` (Linux and Mac OS X) or `/QaxSSE4.1,SSSE3` (Windows). You cannot combine the old style, deprecated options and the new options. For example, you cannot specify `-axSSE4.1,T` (Linux and Mac OS X) or `/QaxSSE4.1,T` (Windows).

Previous values W and K are deprecated. The details on replacements are as follows:

- Mac OS X systems: On these systems, there is no exact replacement for W or K. You can upgrade to the default option `-msse3` (IA-32 architecture) or option `-mssse3` (Intel® 64 architecture).
- Windows and Linux systems: The replacement for W is `-msse2` (Linux) or `/arch:SSE2` (Windows). There is no exact replacement for K. However, on Windows systems, `/QaxK` is interpreted as `/arch:IA32`; on Linux systems, `-axK` is interpreted as `-mia32`. You can also do one of the following:
  - Upgrade to option `-msse2` (Linux) or option `/arch:SSE2` (Windows). This will produce one code path that is specialized for Intel® SSE2. It will not run on earlier processors
  - Specify the two option combination `-mia32 -axSSE2` (Linux) or `/arch:IA32 /QaxSSE2` (Windows). This combination will produce an executable that runs on any processor with IA-32 architecture but with an additional specialized Intel® SSE2 code path.

The `-ax` and `/Qax` options enable additional optimizations not enabled with option `-m` or option `/arch`.

Alternate Options

None

See Also

-
-
- `x`, `Qx`
- `m`
- `arch`

Qchkstk

Enables stack probing when the stack is dynamically expanded at run-time.

IDE Equivalent

Windows: **Run-time > Enable Stack Check Upon Expansion**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Qchkstk`

`/Qchkstk-`

Arguments

None

Default

`/Qchkstk`	Stack probing is enabled when the stack is dynamically expanded at run-time.

Description

This option enables stack probing when the stack is dynamically expanded at run-time.

It instructs the compiler to generate a call to `_chkstk`. The call will probe the requested memory and detect possible stack overflow.

To cancel the call to `_chkstk`, specify `/Qchkstk-`.

Alternate Options

None

common-args, Qcommon-args

See assume.

complex-limited-range, Qcomplex-limited-range

Determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

IDE Equivalent

Windows: **Floating point > Limit COMPLEX Range**

Linux: None

Mac OS X: **Floating point > Limit COMPLEX Range**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-complex-limited-range`

`-no-complex-limited-range`

Windows:

`/Qcomplex-limited-range`

`/Qcomplex-limited-range-`

Arguments

None

Default

`-no-complex-limited-range` or `/Qcomplex-limited-range-`	Basic algebraic expansions of some arithmetic operations involving data of type COMPLEX are disabled.

Description

This option determines whether the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX is enabled.

When the option is enabled, this can cause performance improvements in programs that use a lot of COMPLEX arithmetic. However, values at the extremes of the exponent range may not compute correctly.

Alternate Options

None

cpp, Qcpp

See fpp, Qfpp.

d-lines, Qd-lines

Compiles debug statements.

IDE Equivalent

Windows: **Language > Compile Lines With D in Column 1**

Linux: None

Mac OS X: **Language > Compile Lines With D in Column 1**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-d-lines
-nod-lines
```

Windows:

```
/d-lines
/nod-lines
/Qd-lines
```

Arguments

None

Default

`nod-lines`	Debug lines are treated as comment lines.

Description

This option compiles debug statements. It specifies that lines in fixed-format files that contain a D in column 1 (debug statements) should be treated as source code.

Alternate Options

Linux and Mac OS X: `-DD`

Windows: None

diag, Qdiag

Controls the display of diagnostic information.

IDE Equivalent

Windows: **Diagnostics > Disable Specific Diagnostics** (`/Qdiag-disable:id`)

Diagnostics > Level of Source Code Analysis (`/Qdiag-enable[:sc1,sc2,sc3]`)

Linux: None

Mac OS X: **Diagnostics > Disable Specific Diagnostics** (`-diag-disable id`)

Diagnostics > Level of Source Code Analysis (`-diag-enable [sc1,sc2,sc3]`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-`*`type diag-list`*

Windows:

`/Qdiag-`*`type:diag-list`*

Arguments

type

Is an action to perform on diagnostics. Possible values are:

`enable`	Enables a diagnostic message or a group of messages.
`disable`	Disables a diagnostic message or a group of messages.
`error`	Tells the compiler to change diagnostics to errors.
`warning`	Tells the compiler to change diagnostics to warnings.
`remark`	Tells the compiler to change diagnostics to remarks (comments).

diag-list

Is a diagnostic group or ID value. Possible values are:

`driver`	Specifies diagnostic messages issued by the compiler driver.
`vec`	Specifies diagnostic messages issued by the vectorizer.
`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).
`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
`sc[n]`	Specifies diagnostic messages issued by the Source Checker. *n* can be any of the following: 1, 2, 3. For more details on these values, see below. This value is equivalent to deprecated value sv[n].
`warn`	Specifies diagnostic messages that have a "warning" severity level.
`error`	Specifies diagnostic messages that have an "error" severity level.
`remark`	Specifies diagnostic messages that are remarks or comments.

`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. This diagnostic group is only available on IA-32 architecture and Intel® 64 architecture.
`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each id.
`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each tag.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information. Diagnostic messages are output to stderr unless compiler option `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) is specified.

When *`diag-list`* value "warn" is used with the Source Checker (sc) diagnostics, the following behavior occurs:

- Option `-diag-enable warn` (Linux and Mac OS X) and `/Qdiag-enable:warn` (Windows) enable all Source Checker diagnostics except those that have an "error" severity level. They enable all Source Checker warnings, cautions, and remarks.
- Option `-diag-disable warn` (Linux and Mac OS X) and `/Qdiag-disable:warn` (Windows) disable all Source Checker diagnostics except those that have an "error" severity level. They suppress all Source Checker warnings, cautions, and remarks.

The following table shows more information on values you can specify for diag-list item sc.

diag-list Item	Description
sc[n]	The value of *n* for Source Checker messages can be any of the following: 1 Produces the diagnostics with severity level set to all critical errors. 2 Produces the diagnostics with severity level set to all errors. This is the default if n is not specified. 3 Produces the diagnostics with severity level set to all errors and warnings.

To control the diagnostic information reported by the vectorizer, use the `-vec-report` (Linux and Mac OS X) or `/Qvec-report` (Windows) option.

To control the diagnostic information reported by the auto-parallelizer, use the `-par-report` (Linux and Mac OS X) or `/Qpar-report` (Windows) option.

Alternate Options

enable vec	Linux and Mac OS X: `-vec-report` Windows: `/Qvec-report`
disable vec	Linux and Mac OS X: `-vec-report0` Windows: `/Qvec-report0`
enable par	Linux and Mac OS X: `-par-report` Windows: `/Qpar-report`
disable par	Linux and Mac OS X: `-par-report0` Windows: `/Qpar-report0`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable 117,230,450     ! Linux and Mac OS X systems
/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable vec -diag-warning vec      ! Linux and Mac OS X systems
/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable par     ! Linux and Mac OS X systems

/Qdiag-disable:par    ! Windows systems
```

The following example shows how to produce Source Checker diagnostic messages for all critical errors:

```
-diag-enable sc1      ! Linux and Mac OS X systems

/Qdiag-enable:sc1     ! Windows system
```

The following example shows how to cause Source Checker diagnostics (and default diagnostics) to be sent to a file:

```
-diag-enable sc -diag-file=stat_ver_msg   ! Linux and Mac OS X systems


/Qdiag-enable:sc /Qdiag-file:stat_ver_msg ! Windows systems
```

Note that you need to enable the Source Checker diagnostics before you can send them to a file. In this case, the diagnostics are sent to file stat_ver_msg.diag. If a file name is not specified, the diagnostics are sent to name-of-the-first-source-file.diag.

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error warn,remark    ! Linux and Mac OS X systems

/Qdiag-error:warn,remark   ! Windows systems
```

See Also

-
-
-
-
- diag-dump, Qdiag-dump
- diag-id-numbers, Qdiag-id-numbers
- diag-file, Qdiag-file
- par-report, Qpar-report
- vec-report, Qvec-report

diag-dump, Qdiag-dump

Tells the compiler to print all enabled diagnostic messages and stop compilation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-dump`

Windows:

`/Qdiag-dump`

Arguments

None

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to print all enabled diagnostic messages and stop compilation. The diagnostic messages are output to `stdout`.

This option prints the enabled diagnostics from all possible diagnostics that the compiler can issue, including any default diagnostics.

If `-diag-enable` *`diag-list`* (Linux and Mac OS X) or `/Qdiag-enable` *`diag-list`* (Windows) is specified, the print out will include the *`diag-list`* diagnostics.

Alternate Options

None

Example

The following example adds vectorizer diagnostic messages to the printout of default diagnostics:

```
-diag-enable vec -diag-dump          ! Linux and Mac OS X systems
/Qdiag-enable:vec /Qdiag-dump     ! Windows systems
```

See Also

-
-
- diag, Qdiag

diag, Qdiag

Controls the display of diagnostic information.

IDE Equivalent

Windows: **Diagnostics > Disable Specific Diagnostics** (`/Qdiag-disable:id`)

Diagnostics > Level of Source Code Analysis (`/Qdiag-enable[:sc1,sc2,sc3]`)

Linux: None

Mac OS X: **Diagnostics > Disable Specific Diagnostics** (`-diag-disable id`)

Diagnostics > Level of Source Code Analysis (`-diag-enable [sc1,sc2,sc3]`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-`*`type diag-list`*

Windows:

`/Qdiag-`*`type:diag-list`*

Arguments

type	Is an action to perform on diagnostics. Possible values are:

	`enable`	Enables a diagnostic message or a group of messages.
	`disable`	Disables a diagnostic message or a group of messages.
	`error`	Tells the compiler to change diagnostics to errors.
	`warning`	Tells the compiler to change diagnostics to warnings.
	`remark`	Tells the compiler to change diagnostics to remarks (comments).
`diag-list`	Is a diagnostic group or ID value. Possible values are:	
	`driver`	Specifies diagnostic messages issued by the compiler driver.
	`vec`	Specifies diagnostic messages issued by the vectorizer.
	`par`	Specifies diagnostic messages issued by the auto-parallelizer (parallel optimizer).
	`openmp`	Specifies diagnostic messages issued by the OpenMP* parallelizer.
	`sc[n]`	Specifies diagnostic messages issued by the Source Checker. *`n`* can be any of the following: 1, 2, 3. For more details on these values, see below. This value is equivalent to deprecated value sv[n].
	`warn`	Specifies diagnostic messages that have a "warning" severity level.
	`error`	Specifies diagnostic messages that have an "error" severity level.
	`remark`	Specifies diagnostic messages that are remarks or comments.
	`cpu-dispatch`	Specifies the CPU dispatch remarks for diagnostic messages. These remarks are enabled by default. This diagnostic group is only available on IA-32 architecture and Intel® 64 architecture.

`id[,id,...]`	Specifies the ID number of one or more messages. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each id.
`tag[,tag,...]`	Specifies the mnemonic name of one or more messages. If you specify more than one mnemonic name, they must be separated by commas. There can be no intervening white space between each tag.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option controls the display of diagnostic information. Diagnostic messages are output to stderr unless compiler option `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) is specified.

When *`diag-list`* value "warn" is used with the Source Checker (sc) diagnostics, the following behavior occurs:

- Option `-diag-enable warn` (Linux and Mac OS X) and `/Qdiag-enable:warn` (Windows) enable all Source Checker diagnostics except those that have an "error" severity level. They enable all Source Checker warnings, cautions, and remarks.
- Option `-diag-disable warn` (Linux and Mac OS X) and `/Qdiag-disable:warn` (Windows) disable all Source Checker diagnostics except those that have an "error" severity level. They suppress all Source Checker warnings, cautions, and remarks.

The following table shows more information on values you can specify for diag-list item sc.

diag-list **Item**	**Description**
`sc[n]`	The value of *`n`* for Source Checker messages can be any of the following:
	1 Produces the diagnostics with severity level set to all critical errors.

diag-list Item	Description
2	Produces the diagnostics with severity level set to all errors. This is the default if n is not specified.
3	Produces the diagnostics with severity level set to all errors and warnings.

To control the diagnostic information reported by the vectorizer, use the `-vec-report` (Linux and Mac OS X) or `/Qvec-report` (Windows) option.

To control the diagnostic information reported by the auto-parallelizer, use the `-par-report` (Linux and Mac OS X) or `/Qpar-report` (Windows) option.

Alternate Options

enable vec	Linux and Mac OS X: `-vec-report` Windows: `/Qvec-report`
disable vec	Linux and Mac OS X: `-vec-report0` Windows: `/Qvec-report0`
enable par	Linux and Mac OS X: `-par-report` Windows: `/Qpar-report`
disable par	Linux and Mac OS X: `-par-report0` Windows: `/Qpar-report0`

Example

The following example shows how to enable diagnostic IDs 117, 230 and 450:

```
-diag-enable 117,230,450     ! Linux and Mac OS X systems
/Qdiag-enable:117,230,450    ! Windows systems
```

The following example shows how to change vectorizer diagnostic messages to warnings:

```
-diag-enable vec -diag-warning vec      ! Linux and Mac OS X systems
/Qdiag-enable:vec /Qdiag-warning:vec    ! Windows systems
```

Note that you need to enable the vectorizer diagnostics before you can change them to warnings.

The following example shows how to disable all auto-parallelizer diagnostic messages:

```
-diag-disable par     ! Linux and Mac OS X systems
/Qdiag-disable:par    ! Windows systems
```

The following example shows how to produce Source Checker diagnostic messages for all critical errors:

```
-diag-enable sc1      ! Linux and Mac OS X systems

/Qdiag-enable:sc1     ! Windows system
```

The following example shows how to cause Source Checker diagnostics (and default diagnostics) to be sent to a file:

```
-diag-enable sc -diag-file=stat_ver_msg   ! Linux and Mac OS X systems

/Qdiag-enable:sc /Qdiag-file:stat_ver_msg ! Windows systems
```

Note that you need to enable the Source Checker diagnostics before you can send them to a file. In this case, the diagnostics are sent to file stat_ver_msg.diag. If a file name is not specified, the diagnostics are sent to name-of-the-first-source-file.diag.

The following example shows how to change all diagnostic warnings and remarks to errors:

```
-diag-error warn,remark    ! Linux and Mac OS X systems

/Qdiag-error:warn,remark   ! Windows systems
```

See Also

-
-
-
-
- diag-dump, Qdiag-dump
- diag-id-numbers, Qdiag-id-numbers
- diag-file, Qdiag-file
- par-report, Qpar-report
- vec-report, Qvec-report

diag-enable sc-include, Qdiag-enable:sc-include

Tells a source code analyzer to process include files and source files when issuing diagnostic messages.

IDE Equivalent

Windows: **Diagnostics > Analyze Include Files**

Linux: None

Mac OS X: **Diagnostics > Analyze Include Files**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-diag-enable sc-include`

Windows:

`/Qdiag-enable:sc-include`

Arguments

None

Default

OFF	The compiler issues certain diagnostic messages by default. If the Source Checker is enabled, include files are not analyzed by default.

Description

This option tells a source code analyzer (Source Checker) to process include files and source files when issuing diagnostic messages. Normally, when Source Checker diagnostics are enabled, only source files are analyzed.

To use this option, you must also specify `-diag-enable sc` (Linux and Mac OS X) or `/Qdiag-enable:sc` (Windows) to enable the Source Checker diagnostics, or `-diag-enable sc-parallel` (Linux and Mac OS X) or `/Qdiag-enable:sc-parallel` (Windows) to enable parallel lint.

Alternate Options

Linux and Mac OS X: `-diag-enable sv-include` (this is a deprecated option)

Windows: `/Qdiag-enable:sv-include` (this is a deprecated option)

Example

The following example shows how to cause include files to be analyzed as well as source files:

```
-diag-enable sc -diag-enable sc-include    ! Linux and Mac OS systems
/Qdiag-enable:sc /Qdiag-enable:sc-include  ! Windows systems
```

In the above example, the first compiler option enables Source Checker messages. The second compiler option causes include files referred to by the source file to be analyzed also.

See Also

-
-
- `diag-enable sc-parallel`, `Qdiag-enable:sc-parallel`
- `diag`, `Qdiag`

diag-enable sc-parallel, Qdiag-enable:sc-parallel

Enables analysis of parallelization in source code (parallel lint diagnostics).

IDE Equivalent

Windows: **Diagnostics > Level of Source Code Parallelization Analysis**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-diag-enable sc-parallel[`*n*`]`

Windows:

`/Qdiag-enable:sc-parallel[`*n*`]`

Arguments

n	Is the level of analysis to perform. Possible values are:

1	Produces the diagnostics with severity level set to all critical errors.
2	Tells the compiler to generate a report with the medium level of detail. Produces the diagnostics with severity level set to all errors. This is the default if *n* is not specified.
3	Produces the diagnostics with severity level set to all errors and warnings.

Default

OFF	The compiler does not analyze parallelization in source code.

Description

This option enables analysis of parallelization in source code (parallel lint diagnostics). Currently, this analysis uses OpenMP directives, so this option has no effect unless option `/Qopenmp` (Windows) or option `-openmp` (Linux and Mac OS X) is set.

Parallel lint performs interprocedural source code analysis to identify mistakes when using parallel directives. It reports various problems that are difficult to find, including data dependency and potential deadlocks.

Source Checker diagnostics (enabled by `/Qdiag-enable:sc` on Windows* OS or `-diag-enable sc` on Linux* OS and Mac OS* X) are a superset of parallel lint diagnostics. Therefore, if Source Checker diagnostics are enabled, the parallel lint option is not taken into account.

Alternate Options

None

See Also

-
-
- `diag`, `Qdiag`

diag-error-limit, Qdiag-error-limit

Specifies the maximum number of errors allowed before compilation stops.

IDE Equivalent

Windows: **Compilation Diagnostics > Error Limit**

Linux: None

Mac OS X: **Compiler Diagnostics > Error Limit**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-error-limit`*n*

`-no-diag-error-limit`

Windows:

`/Qdiag-error-limit:`*n*

`/Qdiag-error-limit-`

Arguments

n	Is the maximum number of error-level or fatal-level compiler errors allowed.

Default

30	A maximum of 30 error-level and fatal-level messages are allowed.

Description

This option specifies the maximum number of errors allowed before compilation stops. It indicates the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line.

If you specify `-no-diag-error-limit` (Linux and Mac OS X) or `/Qdiag-error-limit-` (Windows) on the command line, there is no limit on the number of errors that are allowed.

If the maximum number of errors is reached, a warning message is issued and the next file (if any) on the command line is compiled.

Alternate Options

Linux and Mac OS X: `-error-limit` and `-noerror-limit`

Windows: `/error-limit` and `/noerror-limit`

diag-file, Qdiag-file

Causes the results of diagnostic analysis to be output to a file.

IDE Equivalent

Windows: **Diagnostics > Diagnostics File**

Linux: None

Mac OS X: **Diagnostics > Diagnostics File**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-file[=`*`file`*`]`

Windows:

`/Qdiag-file[:`*`file`*`]`

Arguments

`file`	Is the name of the file for output.

Default

OFF	Diagnostic messages are output to stderr.

Description

This option causes the results of diagnostic analysis to be output to a file. The file is placed in the current working directory.

If *file* is specified, the name of the file is *file.diag*. The file can include a file extension; for example, if *file.ext* is specified, the name of the file is *file.ext*.

If *file* is not specified, the name of the file is *name-of-the-first-source-file.diag*. This is also the name of the file if the name specified for file conflicts with a source file name provided in the command line.

NOTE. If you specify `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows) and you also specify `-diag-file-append` (Linux and Mac OS X) or `/Qdiag-file-append` (Windows), the last option specified on the command line takes precedence.

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be output to a file named `my_diagnostics.diag`:

```
-diag-file=my_diagnostics       ! Linux and Mac OS X systems
/Qdiag-file:my_diagnostics      ! Windows systems
```

See Also

-
-
- `diag`, `Qdiag`
- `diag-file-append`, `Qdiag-file-append`

diag-file-append, Qdiag-file-append

Causes the results of diagnostic analysis to be appended to a file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-file-append[=file]`

Windows:

`/Qdiag-file-append[:file]`

Arguments

file	Is the name of the file to be appended to. It can include a path.

Default

OFF	Diagnostic messages are output to `stderr`.

Description

This option causes the results of diagnostic analysis to be appended to a file. If you do not specify a path, the driver will look for *file* in the current working directory.

If *file* is not found, then a new file with that name is created in the current working directory. If the name specified for file conflicts with a source file name provided in the command line. the name of the file is *name-of-the-first-source-file.diag*.

NOTE. If you specify `-diag-file-append` (Linux and Mac OS X) or `/Qdiag-file-append` (Windows) and you also specify `-diag-file` (Linux and Mac OS X) or `/Qdiag-file` (Windows), the last option specified on the command line takes precedence.

Alternate Options

None

Example

The following example shows how to cause diagnostic analysis to be appended to a file named my_diagnostics.txt:

```
-diag-file-append=my_diagnostics.txt       ! Linux and Mac OS X systems
/Qdiag-file-append:my_diagnostics.txt      ! Windows systems
```

See Also

-
-
- diag, Qdiag
- diag-file, Qdiag-file

diag-id-numbers, Qdiag-id-numbers

Determines whether the compiler displays diagnostic messages by using their ID number values.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-diag-id-numbers
-no-diag-id-numbers
```

Windows:

```
/Qdiag-id-numbers
/Qdiag-id-numbers-
```

Arguments

None

Default

`-diag-id-numbers` or `/Qdiag-id-numbers`	The compiler displays diagnostic messages by using their ID number values.

Description

This option determines whether the compiler displays diagnostic messages by using their ID number values. If you specify `-no-diag-id-numbers` (Linux and Mac OS X) or `/Qdiag-id-numbers-` (Windows), mnemonic names are output for driver diagnostics only.

Alternate Options

None

See Also

-
-
- `diag`, `Qdiag`

diag-once, Qdiag-once

Tells the compiler to issue one or more diagnostic messages only once.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-diag-once`*`id`*`[,`*`id`*`,...]`

Windows:

`/Qdiag-once:`*`id`*`[,`*`id`*`,...]`

Arguments

id	Is the ID number of the diagnostic message. If you specify more than one message number, they must be separated by commas. There can be no intervening white space between each *id*.

Default

OFF	The compiler issues certain diagnostic messages by default.

Description

This option tells the compiler to issue one or more diagnostic messages only once.

Alternate Options

None

dps, Qdps

See altparam.

dyncom, Qdyncom

Enables dynamic allocation of common blocks at run time.

IDE Equivalent

Windows: **Data > Dynamic Common Blocks**

Linux: None

Mac OS X: **Data > Dynamic Common Blocks**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-dyncom "common1,common2,..."`

Windows:

```
/Qdyncom "common1,common2,..."
```

Arguments

common1,*common2*,...	Are the names of the common blocks to be dynamically allocated. The list of names must be within quotes.

Default

OFF	Common blocks are not dynamically allocated at run time.

Description

This option enables dynamic allocation of the specified common blocks at run time. For example, to enable dynamic allocation of common blocks a, b, and c at run time, use this syntax:

```
/Qdyncom "a,b,c"    ! on Windows systems

-dyncom "a,b,c"     ! on Linux and Mac OS X systems
```

The following are some limitations that you should be aware of when using this option:

- An entity in a dynamic common cannot be initialized in a DATA statement.
- Only named common blocks can be designated as dynamic COMMON.
- An entity in a dynamic common block must not be used in an EQUIVALENCE expression with an entity in a static common block or a DATA-initialized variable.

Alternate Options

None

See Also

-
-

Building Applications: Allocating Common Blocks

Qextend-source

See extend-source.

fast-transcendentals, Qfast-transcendentals

Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fast-transcendentals`

`-no-fast-transcendentals`

Windows:

`/Qfast-transcendentals`

`/Qfast-transcendentals-`

Default

`-fast-transcendentals` or `/Qfast-transcendentals`	The default depends on the setting of `-fp-model` (Linux and Mac OS X) or `/fp` (Windows). The default is ON if default setting `-fp-model fast` or `/fp:fast` is in effect. However, if a value-safe option such as `-fp-model precise` or `/fp:precise` is specified, the default is OFF.

Description

This option enables the compiler to replace calls to transcendental functions with implementations that may be faster but less precise.

It tells the compiler to perform certain optimizations on transcendental functions, such as replacing individual calls to sine in a loop with a single call to a less precise vectorized sine library routine.

This option has an effect only when specified with one of the following options:

- Windows* OS: `/fp:except` or `/fp:precise`
- Linux* OS and Mac OS* X: `-fp-model except` or `-fp-model precise`

You cannot use this option with option `-fp-model strict` (Linux and Mac OS X) or `/fp:strict` (Windows).

Alternate Options

None

See Also

-
-
- `fp-model, fp`

fma, Qfma

Enables the combining or contraction of floating-point multiplications and add or subtract operations.

IDE Equivalent

Windows: **Floating Point > Contract Floating-Point Operations**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

```
-fma
-no-fma
```

Mac OS X:

None

Windows:

/Qfma

/Qfma-

Arguments

None

Default

-fma or/Qfma	Floating-point multiplications and add/subtract operations are combined. However, if you specify -fp-model strict (Linux) or /fp:strict (Windows), but do not explicitly specify -fma or /Qfma, the default is -no-fma or /Qfma-.

Description

This option enables the combining or contraction of floating-point multiplications and add or subtract operations into a single operation.

Alternate Options

Linux: -IPF-fma (this is a deprecated option)

Windows: /QIPF-fma (this is a deprecated option)

See Also

-
-
- fp-model, fp

Floating-point Operations: Floating-point Options Quick Reference

falign-functions, Qfnalign

Tells the compiler to align functions on an optimal byte boundary.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-falign-functions[=n]`

`-fno-align-functions`

Windows:

`/Qfnalign[:n]`

`/Qfnalign-`

Arguments

n	Is the byte boundary for function alignment. Possible values are 2 or 16.

Default

`-fno-align-functions` or `/Qfnalign-`	The compiler aligns functions on 2-byte boundaries. This is the same as specifying `-falign-functions=2` (Linux and Mac OS X) or `/Qfnalign:2` (Windows).

Description

This option tells the compiler to align functions on an optimal byte boundary. If you do not specify *n*, the compiler aligns the start of functions on 16-byte boundaries.

Alternate Options

None

fnsplit, Qfnsplit

Enables function splitting.

IDE Equivalent

None

Architectures

`/Qfnsplit[-]`: IA-32 architecture, Intel® 64 architecture

`-[no-]fnsplit`: IA-64 architecture

Syntax

Linux:

`-fnsplit`

`-no-fnsplit`

Mac OS X:

`None`

Windows:

`/Qfnsplit`

`/Qfnsplit-`

Arguments

None

Default

`-no-fnsplit` or `/Qfnsplit-`	Function splitting is not enabled unless `-prof-use` (Linux) or `/Qprof-use` (Windows) is also specified.

Description

This option enables function splitting if `-prof-use` (Linux) or `/Qprof-use` (Windows) is also specified. Otherwise, this option has no effect.

It is enabled automatically if you specify `-prof-use` or `/Qprof-use`. If you do not specify one of those options, the default is `-no-fnsplit` (Linux) or `/Qfnsplit-` (Windows), which disables function splitting but leaves function grouping enabled.

To disable function splitting when you use `-prof-use` or `/Qprof-use`, specify `-no-fnsplit` or `/Qfnsplit-`.

Alternate Options

None

See Also

-
-
- Profile-Guided Optimization (PGO) Quick Reference
- Profile an Application

fp-port, Qfp-port

Rounds floating-point results after floating-point operations.

IDE Equivalent

Windows: **Floating-Point > Round Floating-Point Results**

Linux: None

Mac OS X: **Floating-Point > Round Floating-Point Results**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-fp-port
-no-fp-port
```

Windows:

```
/Qfp-port
/Qfp-port-
```

Arguments

None

Default

`-no-fp-port` or `/Qfp-port-`	The default rounding behavior depends on the compiler's code generation decisions and the precision parameters of the operating system.

Description

This option rounds floating-point results after floating-point operations. Rounding to user-specified precision occurs at assignments and type conversions. This has some impact on speed.

The default is to keep results of floating-point operations in higher precision. This provides better performance but less consistent floating-point results.

Alternate Options

None

fp-relaxed, Qfp-relaxed

Enables use of faster but slightly less accurate code sequences for math functions.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-fp-relaxed`

`-no-fp-relaxed`

Mac OS X:

`None`

Windows:

`/Qfp-relaxed`

`/Qfp-relaxed-`

Arguments

None

Default

`-no-fp-relaxed` or `/Qfp-relaxed-`	Default code sequences are used for math functions.

Description

This option enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt. When compared to strict IEEE* precision, this option slightly reduces the accuracy of floating-point calculations performed by these functions, usually limited to the least significant digit.

This option also enables the performance of more aggressive floating-point transformations, which may affect accuracy.

Alternate Options

Linux: `-IPF-fp-relaxed` (this is a deprecated option)

Windows: `/QIPF-fp-relaxed` (this is a deprecated option)

See Also

-
-
- `fp-model, fp`

fp-speculation, Qfp-speculation

Tells the compiler the mode in which to speculate on floating-point operations.

IDE Equivalent

Windows: **Floating Point > Floating-Point Speculation**

Linux: None

Mac OS X: **Floating Point > Floating-Point Speculation**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-fp-speculation=`*mode*

Windows:

`/Qfp-speculation:`*mode*

Arguments

mode	Is the mode for floating-point operations. Possible values are:	
	`fast`	Tells the compiler to speculate on floating-point operations.
	`safe`	Tells the compiler to disable speculation if there is a possibility that the speculation may cause a floating-point exception.
	`strict`	Tells the compiler to disable speculation on floating-point operations.
	`off`	This is the same as specifying strict.

Default

`-fp-speculation=fast` or `/Qfp-speculation:fast`	The compiler speculates on floating-point operations. This is also the behavior when optimizations are enabled. However, if you specify no optimizations (`-O0` on Linux; `/Od` on Windows), the default is `-fp-speculation=safe` (Linux) or `/Qfp-speculation:safe` (Windows).

Description

This option tells the compiler the mode in which to speculate on floating-point operations.

Alternate Options

None

See Also

•
•

Floating-point Operations: Floating-point Options Quick Reference

fp-stack-check, Qfp-stack-check

Tells the compiler to generate extra code after every function call to ensure that the floating-point stack is in the expected state.

IDE Equivalent

Windows: **Floating-Point > Check Floating-point Stack**

Linux: None

Mac OS X: **Floating-Point > Check Floating-point Stack**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-fp-stack-check
```

Windows:

```
/Qfp-stack-check
```

Arguments

None

Default

OFF	There is no checking to ensure that the floating-point (FP) stack is in the expected state.

Description

This option tells the compiler to generate extra code after every function call to ensure that the floating-point (FP) stack is in the expected state.

By default, there is no checking. So when the FP stack overflows, a NaN value is put into FP calculations and the program's results differ. Unfortunately, the overflow point can be far away from the point of the actual bug. This option places code that causes an access violation exception immediately after an incorrect call occurs, thus making it easier to locate these issues.

Alternate Options

None

See Also

•
•

Floating-point Operations:

- Checking the Floating-point Stack State

fpp, Qfpp

Runs the Fortran preprocessor on source files before compilation.

IDE Equivalent

Windows: **Preprocessor > Preprocess Source File**

Linux: None

Mac OS X: Preprocessor > Preprocess Source File

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`fpp[`*`n`*`]`

`fpp[="`*`option`*`"]`

`-nofpp`

Windows:

`/fpp[n]`

`/fpp[:"option"]`

`/nofpp`

`/Qfpp[n]`

`/Qfpp[:"option"]`

Arguments

n	Deprecated. Tells the compiler whether to run the preprocessor or not. Possible values are:
	`0` Tells the compiler not to run the preprocessor.
	`1, 2, or 3` Tells the compiler to run the preprocessor.
option	Is a Fortran preprocessor (fpp) option; for example, "-macro=no", which disables macro expansion. The quotes are required. For a list of fpp options, see *Fortran Preprocessor Options*.

Default

`nofpp`	The Fortran preprocessor is not run on files before compilation.

Description

This option runs the Fortran preprocessor on source files before they are compiled.

If the option is specified with no argument, the compiler runs the preprocessor.

If `0` is specified for *n*, it is equivalent to `nofpp`.Note that argument *n* is deprecated.

We recommend you use option `Qoption,fpp,"option"` to pass fpp options to the Fortran preprocessor.

Alternate Options

Linux and Mac OS X: `-cpp`

Windows: `/Qcpp`

See Also

-
-
- Fortran Preprocessor Options
- Qoption

ftz, Qftz

Flushes denormal results to zero.

IDE Equivalent

Windows: (IA-32 and IA-64 architectures): **Floating Point > Flush Denormal Results to Zero**

(Intel® 64 architecture): None

Linux: None

Mac OS X: **Floating Point > Flush Denormal Results to Zero**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-ftz
```

```
-no-ftz
```

Windows:

```
/Qftz
```

```
/Qftz-
```

Arguments

None

Default

Systems using IA-64 architecture: `-no-ftz` or `/Qftz-` Systems using IA-32 architecture and Intel® 64 architecture: `-ftz` or `/Qftz`	On systems using IA-64 architecture, the compiler lets results gradually underflow. On systems using IA-32 architecture and Intel® 64 architecture, denormal results are flushed to zero.

Description

This option flushes denormal results to zero when the application is in the gradual underflow mode. It may improve performance if the denormal values are not critical to your application's behavior.

This option sets or resets the FTZ and the DAZ hardware flags. If FTZ is ON, denormal results from floating-point calculations will be set to the value zero. If FTZ is OFF, denormal results remain as is. If DAZ is ON, denormal values used as input to floating-point instructions will be treated as zero. If DAZ is OFF, denormal instruction inputs remain as is. Systems using IA-64 architecture have FTZ but not DAZ. Systems using Intel® 64 architecture have both FTZ and DAZ. FTZ and DAZ are not supported on all IA-32 architectures.

When `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows) is used in combination with an SSE-enabling option on systems using IA-32 architecture (for example, `xN` or `QxN`), the compiler will insert code in the main routine to set FTZ and DAZ. When `-ftz` or `/Qftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check. `-no-ftz` (Linux and Mac OS X) or `/Qftz-` (Windows) will prevent the compiler from inserting any code that might set FTZ or DAZ.

This option only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in FTZ/DAZ mode.

Options `-fpe0` and `-fpe1` (Linux and Mac OS X) set `-ftz`. Options `/fpe:0` and `/fpe:1` (Windows) set `/Qftz`.

On systems using IA-64 architecture, optimization option `O3` sets `-ftz` and `/Qftz`; optimization option `O2` sets `-no-ftz` (Linux) and `/Qftz-` (Windows). On systems using IA-32 architecture and Intel® 64 architecture, every optimization option `O` level, except `O0`, sets `-ftz` and `/Qftz`.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by using `-no-ftz` or `/Qftz-` in the command line while still benefiting from the `O3` optimizations.

NOTE. Options `-ftz` and `/Qftz` are performance options. Setting these options does not guarantee that all denormals in a program are flushed to zero. They only cause denormals generated at run time to be flushed to zero.

Alternate Options

None

See Also

-
-
- `x, Qx`

Floating-point Operations: Using the -fpe or /fpe Compiler Option

global-hoist, Qglobal-hoist

Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-global-hoist
-no-global-hoist
```

Windows:

```
/Qglobal-hoist
/Qglobal-hoist-
```

Arguments

None

Default

`-global-hoist` or `/Qglobal-hoist`	Certain optimizations are enabled that can move memory loads.

Description

This option enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source. In most cases, these optimizations are safe and can improve performance.

The `-no-global-hoist` (Linux and Mac OS X) or `/Qglobal-hoist-` (Windows) option is useful for some applications, such as those that use shared or dynamically mapped memory, which can fail if a load is moved too early in the execution stream (for example, before the memory is mapped).

Alternate Options

None

QIA64-fr32

Disables use of high floating-point registers.

IDE Equivalent

Windows: **Floating Point > Disable Use of High Floating-Point Registers**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/QIA64-fr32`

Arguments

None

Default

OFF	Use of high floating-point registers is enabled.

Description

This option disables use of high floating-point registers.

Alternate Options

None

Qlfist

See rcd, Qrcd.

Qimsl

Tells the compiler to link to the IMSL Fortran Numerical Library*(IMSL* library).*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Qimsl`

Arguments

None

Default

OFF	The compiler does not link to the IMSL* library.

Description

This option tells the compiler to link to the IMSL* Fortran Numerical Library* (IMSL* library). This option is applicable for users of editions of the Intel® Fortran Compiler product that include the IMSL* libraries.

Alternate Options

None

inline-debug-info, Qinline-debug-info

Produces enhanced source position information for inlined code. This is a deprecated option.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-inline-debug-info
```

Windows:

```
/Qinline-debug-info
```

Arguments

None

Default

OFF	No enhanced source position information is produced for inlined code.

Description

This option produces enhanced source position information for inlined code. This leads to greater accuracy when reporting the source location of any instruction. It also provides enhanced debug information useful for function call traceback.

To use this option for debugging, you must also specify a debug enabling option, such as `-g` (Linux) or `/debug` (Windows).

Alternate Options

Linux and Mac OS X: `-debug inline-debug-info`

Windows: None

Qinline-dllimport

Determines whether dllimport functions are inlined.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Qinline-dllimport`

`/Qinline-dllimport-`

Arguments

None

Default

`/Qinline-dllimport`	The dllimport functions are inlined.

Description

This option determines whether dllimport functions are inlined. To disable dllimport functions from being inlined, specify `/Qinline-dllimport-`.

Alternate Options

None

inline-factor, Qinline-factor

Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-factor=`*n*

`-no-inline-factor`

Windows:

`/Qinline-factor=`*n*

`/Qinline-factor-`

Arguments

n	Is a positive integer specifying the percentage value. The default value is 100 (a factor of 1).

Default

`-no-inline-factor` or `/Qinline-factor-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the percentage multiplier that should be applied to all inlining options that define upper limits:

- `-inline-max-size` and `/Qinline-max-size`
- `-inline-max-total-size` and `/Qinline-max-total-size`
- `-inline-max-per-routine` and `/Qinline-max-per-routine`
- `-inline-max-per-compile` and `/Qinline-max-per-compile`

This option takes the default value for each of the above options and multiplies it by *n* divided by 100. For example, if 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2. This option is useful if you do not want to individually increase each option limit.

If you specify `-no-inline-factor` (Linux and Mac OS X) or `/Qinline-factor-` (Windows), the following occurs:

- Every function is considered to be a small or medium function; there are no large functions.
- There is no limit to the size a routine may grow when inline expansion is performed.
- There is no limit to the number of times some routine may be inlined into a particular routine.
- There is no limit to the number of times inlining can be applied to a compilation unit.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase default limits, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- inline-max-size, Qinline-max-size
- inline-max-total-size, Qinline-max-total-size
- inline-max-per-routine, Qinline-max-per-routine
- inline-max-per-compile, Qinline-max-per-compile
- opt-report, Qopt-report
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-forceinline, Qinline-forceinline

Specifies that an inline routine should be inlined whenever the compiler can do so.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-inline-forceinline
```

Windows:

```
/Qinline-forceinline
```

Default

OFF	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies that a inline routine should be inlined whenever the compiler can do so. This causes the routines marked with an inline keyword or directive to be treated as if they were "forceinline".

NOTE. Because C++ member functions whose definitions are included in the class declaration are considered inline functions by default, using this option will also make these member functions "forceinline" functions.

The "forceinline" condition can also be specified by using the directive cDEC$ ATTRIBUTES FORCEINLINE.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS) or `/Qopt-report` (Windows).

CAUTION. When you use this option to change the meaning of inline to "forceinline", the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-per-compile, Qinline-max-per-compile

Specifies the maximum number of times inlining may be applied to an entire compilation unit.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-per-compile=`*n*

`-no-inline-max-per-compile`

Windows:

`/Qinline-max-per-compile=`*n*

`/Qinline-max-per-compile-`

Arguments

n	Is a positive integer that specifies the number of times inlining may be applied.

Default

`-no-inline-max-per-compile` or `/Qinline-max-per-compile-`	The compiler uses default heuristics for inline routine expansion.

Description

This option the maximum number of times inlining may be applied to an entire compilation unit. It limits the number of times that inlining can be applied.

For compilations using Interprocedural Optimizations (IPO), the entire compilation is a compilation unit. For other compilations, a compilation unit is a file.

If you specify `-no-inline-max-per-compile` (Linux and Mac OS X) or `/Qinline-max-per-compile-` (Windows), there is no limit to the number of times inlining may be applied to a compilation unit.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-per-routine, Qinline-max-per-routine

Specifies the maximum number of times the inliner may inline into a particular routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-per-routine=`*n*

`-no-inline-max-per-routine`

Windows:

`/Qinline-max-per-routine=`*n*

`/Qinline-max-per-routine-`

Arguments

n	Is a positive integer that specifies the maximum number of times the inliner may inline into a particular routine.

Default

`-no-inline-max-per-routine` or `/Qinline-max-per-routine-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the maximum number of times the inliner may inline into a particular routine. It limits the number of times that inlining can be applied to any routine.

If you specify `-no-inline-max-per-routine` (Linux and Mac OS X) or `/Qinline-max-per-routine-` (Windows), there is no limit to the number of times some routine may be inlined into a particular routine.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-size, Qinline-max-size

Specifies the lower limit for the size of what the inliner considers to be a large routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-size=`*n*

`-no-inline-max-size`

Windows:

`/Qinline-max-size=`*n*

`/Qinline-max-size-`

Arguments

n	Is a positive integer that specifies the minimum size of what the inliner considers to be a large routine.

Default

`-no-inline-max-size` or `/Qinline-max-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the lower limit for the size of what the inliner considers to be a large routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be medium and large-size routines.

The inliner prefers to inline small routines. It has a preference against inlining large routines. So, any large routine is highly unlikely to be inlined.

If you specify `-no-inline-max-size` (Linux and Mac OS X) or `/Qinline-max-size-` (Windows), there are no large routines. Every routine is either a small or medium routine.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-min-size`, `Qinline-min-size`
- `inline-factor`, `Qinline-factor`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-max-total-size, Qinline-max-total-size

Specifies how much larger a routine can normally grow when inline expansion is performed.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-max-total-size=`*n*

`-no-inline-max-total-size`

Windows:

`/Qinline-max-total-size=`*n*

`/Qinline-max-total-size-`

Arguments

n	Is a positive integer that specifies the permitted increase in the routine's size when inline expansion is performed.

Default

`-no-inline-max-total-size` or `/Qinline-max-total-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies how much larger a routine can normally grow when inline expansion is performed. It limits the potential size of the routine. For example, if 2000 is specified for *n*, the size of any routine will normally not increase by more than 2000.

If you specify `-no-inline-max-total-size` (Linux and Mac OS X) or `/Qinline-max-total-size-` (Windows), there is no limit to the size a routine may grow when inline expansion is performed.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- inline-factor, Qinline-factor
- opt-report, Qopt-report
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

inline-min-size, Qinline-min-size

Specifies the upper limit for the size of what the inliner considers to be a small routine.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-inline-min-size=`*n*

`-no-inline-min-size`

Windows:

`/Qinline-min-size=`*n*

`/Qinline-min-size-`

Arguments

n	Is a positive integer that specifies the maximum size of what the inliner considers to be a small routine.

Default

`-no-inline-min-size` or `/Qinline-min-size-`	The compiler uses default heuristics for inline routine expansion.

Description

This option specifies the upper limit for the size of what the inliner considers to be a small routine (a function or subroutine). The inliner classifies routines as small, medium, or large. This option specifies the boundary between what the inliner considers to be small and medium-size routines.

The inliner has a preference to inline small routines. So, when a routine is smaller than or equal to the specified size, it is very likely to be inlined.

If you specify `-no-inline-min-size` (Linux and Mac OS X) or `/Qinline-min-size-` (Windows), there is no limit to the size of small routines. Every routine is a small routine; there are no medium or large routines.

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

To see compiler values for important inlining limits, specify compiler option `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

CAUTION. When you use this option to increase the default limit, the compiler may do so much additional inlining that it runs out of memory and terminates with an "out of memory" message.

Alternate Options

None

See Also

-
-
- `inline-min-size`, `Qinline-min-size`
- `opt-report`, `Qopt-report`
- Developer Directed Inline Expansion of User Functions
- Compiler Directed Inline Expansion of User Functions

Qinstall

Specifies the root directory where the compiler installation was performed.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Qinstall`*`dir`*

Windows:

`None`

Arguments

`dir`	Is the root directory where the installation was performed.

Default

OFF	The default root directory for compiler installation is searched for the compiler.

Description

This option specifies the root directory where the compiler installation was performed. It is useful if you want to use a different compiler or if you did not use the ifortvars shell script to set your environment variables.

Alternate Options

None

minstruction, Qinstruction

Determines whether MOVBE instructions are generated for Intel processors.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-minstruction=[no]movbe`

Windows:

`/Qinstruction:[no]movbe`

Arguments

None

Default

`-minstruction=nomovbe` or `/Qinstruction:nomovbe` The compiler does not generate MOVBE instructions for Intel® Atom™ processors.

Description

This option determines whether MOVBE instructions are generated for Intel processors. To use this option, you must also specify `-xSSE3_ATOM` (Linux and Mac OS X) or `/QxSSE3_ATOM` (Windows).

If `-minstruction=movbe` or `/Qinstruction:movbe` is specified, the following occurs:

- MOVBE instructions are generated that are specific to the Intel® Atom™ processor.
- The options are ON by default when `-xSSE3_ATOM` or `/QxSSE3_ATOM` is specified.
- Generated executables can only be run on Intel® Atom™ processors or processors that support Intel® Streaming SIMD Extensions 3 (Intel® SSE3) and MOVBE.

If -minstruction=nomovbe or /Qinstruction:nomovbe is specified, the following occurs:

- The compiler optimizes code for the Intel® Atom™ processor, but it does not generate MOVBE instructions.
- Generated executables can be run on non-Intel® Atom™ processors that support Intel® SSE3.

Alternate Options

None

See Also

-
-
- x, Qx

finstrument-functions, Qinstrument-functions

Determines whether routine entry and exit points are instrumented.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-finstrument-functions
-fno-instrument-functions
```

Windows:

```
/Qinstrument-functions
/Qinstrument-functions-
```

Arguments

None

Default

`-fno-instrument-functions` or `/Qinstrument-functions-`	Routine entry and exit points are not instrumented.

Description

This option determines whether routine entry and exit points are instrumented. It may increase execution time.

The following profiling functions are called with the address of the current routine and the address of where the routine was called (its "call site"):

- This function is called upon routine entry:
  - On IA-32 architecture and Intel® 64 architecture:

    ```
    void __cyg_profile_func_enter (void *this_fn,
    void *call_site);
    ```

  - On IA-64 architecture:

    ```
    void __cyg_profile_func_enter (void **this_fn,
    void *call_site);
    ```

- This function is called upon routine exit:
  - On IA-32 architecture and Intel® 64 architecture:

    ```
    void __cyg_profile_func_exit (void *this_fn,
    void *call_site);
    ```

  - On IA-64 architecture:

    ```
    void __cyg_profile_func_exit (void **this_fn,
    void *call_site);
    ```

On IA-64 architecture, the additional de-reference of the function pointer argument is required to obtain the routine entry point contained in the first word of the routine descriptor for indirect routine calls. The descriptor is documented in the Intel® Itanium® Software Conventions and Runtime Architecture Guide, section 8.4.2. You can find this design guide at web site http://www.intel.com.

These functions can be used to gather more information, such as profiling information or timing information. Note that it is the user's responsibility to provide these profiling functions.

If you specify `-finstrument-functions` (Linux and Mac OS X) or `/Qinstrument-functions` (Windows), routine inlining is disabled. If you specify `-fno-instrument-functions` or `/Qinstrument-functions-`, inlining is not disabled.

This option is provided for compatibility with gcc.

Alternate Options

None

ip, Qip

Determines whether additional interprocedural optimizations for single-file compilation are enabled.

IDE Equivalent

Windows: **Optimization > Interprocedural Optimization**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ip`

`-no-ip`

Windows:

`/Qip`

`/Qip-`

Arguments

None

Default

OFF	Some limited interprocedural optimizations occur, including inline function expansion for calls to functions defined within the current source file. These optimizations are a subset of full intra-file interprocedural optimizations. Note that this setting is not the same as `-no-ip` (Linux and Mac OS X) or `/Qip-` (Windows).

Description

This option determines whether additional interprocedural optimizations for single-file compilation are enabled.

Options `-ip` (Linux and Mac OS X) and `/Qip` (Windows) enable additional interprocedural optimizations for single-file compilation.

Options `-no-ip` (Linux and Mac OS X) and `/Qip-` (Windows) may not disable inlining. To ensure that inlining of user-defined functions is disabled, specify `-inline-level=0`or `-fno-inline` (Linux and Mac OS X), or specify `/Ob0` (Windows).

Alternate Options

None

See Also

-
-
- `finline-functions`

ip-no-inlining, Qip-no-inlining

Disables full and partial inlining enabled by interprocedural optimization options.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ip-no-inlining`

Windows:

`/Qip-no-inlining`

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables full and partial inlining enabled by the following interprocedural optimization options:

- On Linux and Mac OS X systems: `-ip` or `-ipo`
- On Windows systems: `/Qip`, `/Qipo`, or `/Ob2`

It has no effect on other interprocedural optimizations.

On Windows systems, this option also has no effect on user-directed inlining specified by option `/Ob1`.

Alternate Options

None

ip-no-pinlining, Qip-no-pinlining

Disables partial inlining enabled by interprocedural optimization options.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-ip-no-pinlining`

Windows:

`/Qip-no-pinlining`

Arguments

None

Default

OFF	Inlining enabled by interprocedural optimization options is performed.

Description

This option disables partial inlining enabled by the following interprocedural optimization options:

- On Linux and Mac OS X systems: `-ip` or `-ipo`
- On Windows systems: `/Qip` or `/Qipo`

It has no effect on other interprocedural optimizations.

Alternate Options

None

IPF-flt-eval-method0, QIPF-flt-eval-method0

Tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program. This is a deprecated option.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-IPF-flt-eval-method0`

Mac OS X:

`None`

Windows:

`/QIPF-flt-eval-method0`

Arguments

None

Default

OFF	Expressions involving floating-point operands are evaluated by default rules.

Description

This option tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program.

By default, intermediate floating-point expressions are maintained in higher precision.

The recommended method to control the semantics of floating-point calculations is to use option `-fp-model` (Linux) or `/fp` (Windows).

Instead of using `-IPF-flt-eval-method0` (Linux) or `/QIPF-flt-eval-method0` (Windows), you can use `-fp-model source` (Linux) or `/fp:source` (Windows).

Alternate Options

None

See Also

-
-

- fp-model, fp

IPF-fltacc, QIPF-fltacc

Disables optimizations that affect floating-point accuracy. This is a deprecated option.

IDE Equivalent

Windows: **Floating Point > Floating-Point Accuracy**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

`-IPF-fltacc`

`-no-IPF-fltacc`

Mac OS X:

`None`

Windows:

`/QIPF-fltacc`

`/QIPF-fltacc-`

Arguments

None

Default

`-no-IPF-fltacc` or `/QIPF-fltacc-`	Optimizations are enabled that affect floating-point accuracy.

Description

This option disables optimizations that affect floating-point accuracy.

If the default setting is used, the compiler may apply optimizations that reduce floating-point accuracy.

You can use this option to improve floating-point accuracy, but at the cost of disabling some optimizations.

The recommended method to control the semantics of floating-point calculations is to use option `-fp-model` (Linux) or `/fp` (Windows).

Instead of using `-IPF-fltacc` (Linux) or `/QIPF-fltacc` (Windows), you can use `-fp-model precise` (Linux) or `/fp:precise` (Windows).

Instead of using `-no-IPF-fltacc` (Linux) or `/QIPF-fltacc-` (Windows), you can use `-fp-model fast` (Linux) or `/fp:fast` (Windows).

Alternate Options

None

See Also

-
-
- `fp-model, fp`

IPF-fma, QIPF-fma

See fma, Qfma.

IPF-fp-relaxed, QIPF-fp-relaxed

See fp-relaxed, Qfp-relaxed.

ipo, Qipo

Enables interprocedural optimization between files.

IDE Equivalent

Windows: **Optimization > Interprocedural Optimization**

General > Whole Program Optimization

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo[n]`

Windows:

`/Qipo[n]`

Arguments

n	Is an optional integer that specifies the number of object files the compiler should create. The integer must be greater than or equal to 0.

Default

OFF	Multifile interprocedural optimization is not enabled.

Description

This option enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or Whole Program Optimization (WPO).

When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

You cannot specify the names for the files that are created.

If *n* is 0, the compiler decides whether to create one or more object files based on an estimate of the size of the application. It generates one object file for small applications, and two or more object files for large applications.

If *n* is greater than 0, the compiler generates *n* object files, unless *n* exceeds the number of source files (*m*), in which case the compiler generates only *m* object files.

If you do not specify *n*, the default is 0.

Alternate Options

None

See Also

•
•

Optimizing Applications:

Interprocedural Optimization (IPO) Quick Reference

Interprocedural Optimization (IPO) Overview

Using IPO

ipo-c, Qipo-c

Tells the compiler to optimize across multiple files and generate a single object file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-c`

Windows:

`/Qipo-c`

Arguments

None

Default

OFF	The compiler does not generate a multifile object file.

Description

This option tells the compiler to optimize across multiple files and generate a single object file (named ipo_out.o on Linux and Mac OS X systems; ipo_out.obj on Windows systems).

It performs the same optimizations as `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), but compilation stops before the final link stage, leaving an optimized object file that can be used in further link steps.

Alternate Options

None

See Also

-
-
- `ipo`, `Qipo`

ipo-jobs, Qipo-jobs

Specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-jobs`*n*

Windows:

`/Qipo-jobs:`*n*

Arguments

n	Is the number of commands (jobs) to run simultaneously. The number must be greater than or equal to 1.

Default

`-ipo-jobs1` or `/Qipo-jobs:1`	One command (job) is executed in an interprocedural optimization parallel build.

Description

This option specifies the number of commands (jobs) to be executed simultaneously during the link phase of Interprocedural Optimization (IPO). It should only be used if the link-time compilation is generating more than one object. In this case, each object is generated by a separate compilation, which can be done in parallel.

This option can be affected by the following compiler options:

- `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) when applications are large enough that the compiler decides to generate multiple object files.
- `-ipo`*n* (Linux and Mac OS X) or `/Qipo`*n* (Windows) when *n* is greater than 1.
- `-ipo-separate` (Linux) or `/Qipo-separate` (Windows)

CAUTION. Be careful when using this option. On a multi-processor system with lots of memory, it can speed application build time. However, if *n* is greater than the number of processors, or if there is not enough memory to avoid thrashing, this option can increase application build time.

Alternate Options

None

See Also

-
-
- `ipo`, `Qipo`
- `ipo-separate`, `Qipo-separate`

ipo-S, Qipo-S

Tells the compiler to optimize across multiple files and generate a single assembly file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ipo-S`

Windows:

`/Qipo-S`

Arguments

None

Default

OFF	The compiler does not generate a multifile assembly file.

Description

This option tells the compiler to optimize across multiple files and generate a single assembly file (named ipo_out.s on Linux and Mac OS X systems; ipo_out.asm on Windows systems).

It performs the same optimizations as `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), but compilation stops before the final link stage, leaving an optimized assembly file that can be used in further link steps.

Alternate Options

None

See Also

•

-
- ipo, Qipo

ipo-separate, Qipo-separate

Tells the compiler to generate one object file for every source file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-ipo-separate`

Mac OS X:

`None`

Windows:

`/Qipo-separate`

Arguments

None

Default

OFF	The compiler decides whether to create one or more object files.

Description

This option tells the compiler to generate one object file for every source file. It overrides any `-ipo` (Linux) or `/Qipo` (Windows) specification.

Alternate Options

None

See Also

-
-
- ipo, Qipo

ivdep-parallel, Qivdep-parallel

Tells the compiler that there is no loop-carried memory dependency in the loop following an IVDEP directive.

IDE Equivalent

Windows: **Optimization > IVDEP Directive Memory Dependency**

Linux: None

Mac OS X: None

Architectures

IA-64 architecture

Syntax

Linux:

`-ivdep-parallel`

Mac OS X:

`None`

Windows:

`/Qivdep-parallel`

Arguments

None

Default

OFF	There may be loop-carried memory dependency in a loop that follows an IVDEP directive.

Description

This option tells the compiler that there is no loop-carried memory dependency in the loop following an IVDEP There may be loop-carried memory dependency in a loop that follows an IVDEP directive.
This has the same effect as specifying the IVDEP:LOOP directive.

Alternate Options

None

See Also

•
•

Optimizing Applications: Absence of Loop-carried Memory Dependency with IVDEP Directive

fkeep-static-consts, Qkeep-static-consts

Tells the compiler to preserve allocation of variables that are not referenced in the source.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-fkeep-static-consts
-fno-keep-static-consts
```

Windows:

```
/Qkeep-static-consts
/Qkeep-static-consts-
```

Arguments

None

Default

`-fno-keep-static-consts` or `/Qkeep-static-consts-`	If a variable is never referenced in a routine, the variable is discarded unless optimizations are disabled by option `-O0` (Linux and Mac OS X) or `/Od` (Windows).

Description

This option tells the compiler to preserve allocation of variables that are not referenced in the source.

The negated form can be useful when optimizations are enabled to reduce the memory usage of static data.

Alternate Options

None

Qlocation

Specifies the directory for supporting tools.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Qlocation,`*`string`*`,`*`dir`*

Windows:

`/Qlocation,`*`string`*`,`*`dir`*

Arguments

`string`	Is the name of the tool.
`dir`	Is the directory (path) where the tool is located.

Default

OFF	The compiler looks for tools in a default area.

Description

This option specifies the directory for supporting tools.

string can be any of the following:

- f - Indicates the Intel Fortran compiler.
- fpp (or cpp) - Indicates the Intel Fortran preprocessor.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux and Mac OS X systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler.
  - ld - Indicates the loader.
  - gld - Indicates the GNU loader.
  - lib - Indicates an additional library.
  - crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

Alternate Options

None

Example

The following command provides the path for the fpp tool:

```
ifort -Qlocation,fpp,/usr/preproc myprog.f
```

See Also

-
- `Qoption`

lowercase, Qlowercase

See names.

map-opts, Qmap-opts

Maps one or more compiler options to their equivalent on a different operating system.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-map-opts`

Mac OS X:

`None`

Windows:

`/Qmap-opts`

Arguments

None

Default

OFF	No platform mappings are performed.

Description

This option maps one or more compiler options to their equivalent on a different operating system. The result is output to `stdout`.

On Windows systems, the options you provide are presumed to be Windows options, so the options that are output to `stdout` will be Linux equivalents.
On Linux systems, the options you provide are presumed to be Linux options, so the options that are output to `stdout` will be Windows equivalents.

The tool can be invoked from the compiler command line or it can be used directly.

No compilation is performed when the option mapping tool is used.

This option is useful if you have both compilers and want to convert scripts or makefiles.

NOTE. Compiler options are mapped to their equivalent on the architecture you are using.
For example, if you are using a processor with IA-32 architecture, you will only see equivalent options that are available on processors with IA-32 architecture.

Alternate Options

None

Example

The following command line invokes the option mapping tool, which maps the Linux options to Windows-based options, and then outputs the results to `stdout`:

```
ifort -map-opts -xP -O2
```

The following command line invokes the option mapping tool, which maps the Windows options to Linux-based options, and then outputs the results to `stdout`:

```
ifort /Qmap-opts /QxP /O2
```

See Also

-
-

Building Applications: Using the Option Mapping Tool

mkl, Qmkl

Tells the compiler to link to certain parts of the Intel® Math Kernel Library (Intel® MKL).

IDE Equivalent

Windows: **Libraries > Use Intel(R) Math Kernel Library**

Linux: None

Mac OS X: **Libraries > Use Inte(R)l Math Kernel Library**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-mkl[=`*`lib`*`]`

Windows:

`/Qmkl[:`*`lib`*`]`

Arguments

`lib`

Indicates the part of the library that the compiler should link to. Possible values are:

`parallel`
Tells the compiler to link using the threaded part of the Intel® MKL. This is the default if the option is specified with no *`lib`*.

`sequential`
Tells the compiler to link using the non-threaded part of the Intel® MKL.

`cluster`
Tells the compiler to link using the cluster part and the sequential part of the Intel® MKL.

Default

OFF
The compiler does not link to the Intel® MKL.

Description

This option tells the compiler to link to certain parts of the Intel® Math Kernel Library (Intel® MKL).

Alternate Options

None

no-bss-init, Qnobss-init

Tells the compiler to place in the DATA section any variables explicitly initialized with zeros.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-no-bss-init
```

Windows:

```
/Qnobss-init
```

Arguments

None

Default

OFF	Variables explicitly initialized with zeros are placed in the BSS section.

Description

This option tells the compiler to place in the DATA section any variables explicitly initialized with zeros.

Alternate Options

Linux and Mac OS X: `-nobss-init` (this is a deprecated option)

Windows: None

onetrip, Qonetrip

Tells the compiler to follow the FORTRAN 66 Standard and execute DO loops at least once.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-onetrip`

Windows:

`/Qonetrip`

Arguments

None

Default

OFF	The compiler applies the current Fortran Standard semantics, which allows zero-trip DO loops.

Description

This option tells the compiler to follow the FORTRAN 66 Standard and execute DO loops at least once.

Alternate Options

Linux and Mac OS X: -1

Windows: /1

openmp, Qopenmp

Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.*

IDE Equivalent

Windows: **Language > Process OpenMP Directives**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp`

Windows:

`/Qopenmp`

Arguments

None

Default

OFF	No OpenMP multi-threaded code is generated by the compiler.

Description

This option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems.

If you use this option, multithreaded libraries are used, but option `fpp` is not automatically invoked.

This option sets option `automatic`.

This option works with any optimization level. Specifying no optimization (`-O0` on Linux or `/Od` on Windows) helps to debug OpenMP applications.

NOTE. On Mac OS X systems, when you enable OpenMP*, you must also set the DYLD_LIBRARY_PATH environment variable within Xcode or an error will be displayed.

Alternate Options

None

See Also

-
-
- `openmp-stubs`, `Qopenmp-stubs`

openmp-lib, Qopenmp-lib

Lets you specify an OpenMP run-time library to use for linking.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-lib` *`type`*

Mac OS X:

`None`

Windows:

`/Qopenmp-lib:`*`type`*

Arguments

`type`	Specifies the type of library to use; it implies compatibility levels. Possible values are:

`legacy`	Tells the compiler to use the legacy OpenMP* run-time library (libguide). This setting does not provide compatibility with object files created using other compilers. This is a deprecated option.
`compat`	Tells the compiler to use the compatibility OpenMP* run-time library (libiomp). This setting provides compatibility with object files created using Microsoft* and GNU* compilers.

Default

`-openmp-lib compat` or `/Qopenmp-lib:compat`	The compiler uses the compatibility OpenMP* run-time library (libiomp).

Description

This option lets you specify an OpenMP* run-time library to use for linking.

The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

The compatibility OpenMP run-time library is compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) and GNU OpenMP run-time library (libgomp).

To use the compatibility OpenMP run-time library, compile and link your application using the `-openmp-lib compat` (Linux) or `/Qopenmp-lib:compat` (Windows) option. To use this option, you must also specify one of the following compiler options:

- Linux OS: `-openmp`, `-openmp-profile`, or `-openmp-stubs`
- Windows OS: `/Qopenmp`, `/Qopenmp-profile`, or `/Qopenmp-stubs`

On Windows* systems, the compatibility OpenMP* run-time library lets you combine OpenMP* object files compiled with the Microsoft* C/C++ compiler with OpenMP* object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

On Linux* systems, the compatibility Intel OpenMP* run-time library lets you combine OpenMP* object files compiled with the GNU* gcc or gfortran compilers with similar OpenMP* object files compiled with the Intel C/C++ or Fortran compilers. The linking phase results in a single, coherent copy of the run-time library.

You cannot link object files generated by the Intel® Fortran compiler to object files compiled by the GNU Fortran compiler, regardless of the presence or absence of the `-openmp` (Linux) or `/Qopenmp` (Windows) compiler option. This is because the Fortran run-time libraries are incompatible.

NOTE. The compatibility OpenMP run-time library is not compatible with object files created using versions of the Intel compiler earlier than 10.0.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`
- `openmp-stubs`, `Qopenmp-stubs`
- `openmp-profile`, `Qopenmp-profile`

openmp-link, Qopenmp-link

Controls whether the compiler links to static or dynamic OpenMP run-time libraries.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-link` *`library`*

Windows:

`/Qopenmp-link:`*`library`*

Arguments

library	Specifies the OpenMP library to use. Possible values are:	
	`static`	Tells the compiler to link to static OpenMP run-time libraries.
	`dynamic`	Tells the compiler to link to dynamic OpenMP run-time libraries.

Default

`-openmp-link dynamic` or `/Qopenmp-link:dynamic`	The compiler links to dynamic OpenMP run-time libraries. However, if option `static` is specified, the compiler links to static OpenMP run-time libraries.

Description

This option controls whether the compiler links to static or dynamic OpenMP run-time libraries.

To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify `-openmp-link static` (Linux and Mac OS X) or `/Qopenmp-link:static` (Windows). However, we strongly recommend you use the default setting, `-openmp-link dynamic` (Linux and Mac OS X) or `/Qopenmp-link:dynamic` (Windows).

NOTE. Compiler options `-static-intel` and `-shared-intel` (Linux and Mac OS X) have no effect on which OpenMP run-time library is linked.

Alternate Options

None

openmp-profile, Qopenmp-profile

Enables analysis of OpenMP applications if Intel® Thread Profiler is installed.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-profile`

Mac OS X:

`None`

Windows:

`/Qopenmp-profile`

Arguments

None

Default

OFF	OpenMP applications are not analyzed.

Description

This option enables analysis of OpenMP* applications. To use this option, you must have previously installed Intel® Thread Profiler, which is one of the Intel® Threading Analysis Tools.

This option can adversely affect performance because of the additional profiling and error checking invoked to enable compatibility with the threading tools. Do not use this option unless you plan to use the Intel® Thread Profiler.

For more information about Intel® Thread Profiler, open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

openmp-report, Qopenmp-report

Controls the OpenMP parallelizer's level of diagnostic messages.*

IDE Equivalent

Windows: **Compilation Diagnostics > OpenMP Diagnostic Level**

Linux: None

Mac OS X: **Compiler Diagnostics > OpenMP Report**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-report[n]`

Windows:

`/Qopenmp-report[n]`

Arguments

n	Is the level of diagnostic messages to display. Possible values are:	
	`0`	No diagnostic messages are displayed.
	`1`	Diagnostic messages are displayed indicating loops, regions, and sections successfully parallelized.
	`2`	The same diagnostic messages are displayed as specified by openmp_report1 plus diagnostic messages indicating successful handling of MASTER constructs, SINGLE constructs, CRITICAL constructs, ORDERED constructs, ATOMIC directives, and so forth.

Default

`-openmp-report1` or `/Qopenmp-report1`	If you do not specify *n*, the compiler displays diagnostic messages indicating loops, regions, and sections successfully parallelized. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the OpenMP* parallelizer's level of diagnostic messages. To use this option, you must also specify `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows).

If this option is specified on the command line, the report is sent to `stdout`.

On Windows systems, if this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`

Optimizing Applications:

Using Parallelism

OpenMP* Report

openmp-stubs, Qopenmp-stubs

Enables compilation of OpenMP programs in sequential mode.

IDE Equivalent

Windows: **Language > Process OpenMP Directives**

Linux: None

Mac OS X: **Language > Process OpenMP Directives**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-openmp-stubs`

Windows:

`/Qopenmp-stubs`

Arguments

None

Default

OFF	The library of OpenMP function stubs is not linked.

Description

This option enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.

Alternate Options

None

See Also

-
-
- `openmp`, `Qopenmp`

openmp-threadprivate, Qopenmp-threadprivate

Lets you specify an OpenMP threadprivate implementation.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-openmp-threadprivate` *type*

Mac OS X:

`None`

Windows:

`/Qopenmp-threadprivate:`*type*

Arguments

type	Specifies the type of threadprivate implementation. Possible values are:	
	`legacy`	Tells the compiler to use the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler. This setting does not provide compatibility with the implementation used by other compilers.
	`compat`	Tells the compiler to use the compatibility OpenMP* threadprivate implementation based on applying the thread-local attribute to each threadprivate variable. This setting provides compatibility with the implementation provided by the Microsoft* and GNU* compilers.

Default

`-openmp-threadprivate legacy` or `/Qopenmp-threadprivate:legacy`	The compiler uses the legacy OpenMP* threadprivate implementation used in the previous releases of the Intel® compiler.

Description

This option lets you specify an OpenMP* threadprivate implementation.

The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

To use this option, you must also specify one of the following compiler options:

- Linux OS: `-openmp`, `-openmp-profile`, or `-openmp-stubs`
- Windows OS: `/Qopenmp`, `/Qopenmp-profile`, or `/Qopenmp-stubs`

The value specified for this option is independent of the value used for option `-openmp-lib` (Linux) or `/Qopenmp-lib` (Windows).

Alternate Options

None

opt-block-factor, Qopt-block-factor

Lets you specify a loop blocking factor.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-block-factor=`*n*

Windows:

`/Qopt-block-factor:`*n*

Arguments

n	Is the blocking factor. It must be an integer. The compiler may ignore the blocking factor if the value is 0 or 1.

Default

OFF	The compiler uses default heuristics for loop blocking.

Description

This option lets you specify a loop blocking factor.

Alternate Options

None

opt-jump-tables, Qopt-jump-tables

Enables or disables generation of jump tables for switch statements.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-jump-tables=`*`keyword`*

`-no-opt-jump-tables`

Windows:

`/Qopt-jump-tables:`*`keyword`*

`/Qopt-jump-tables-`

Arguments

`keyword`	Is the instruction for generating jump tables. Possible values are:	
	`never`	Tells the compiler to never generate jump tables. All switch statements are implemented as chains of if-then-elses.

	This is the same as specifying `-no-opt-jump-tables` (Linux and Mac OS) or `/Qopt-jump-tables-` (Windows).
`default`	The compiler uses default heuristics to determine when to generate jump tables.
`large`	Tells the compiler to generate jump tables up to a certain pre-defined size (64K entries).
`n`	Must be an integer. Tells the compiler to generate jump tables up to *n* entries in size.

Default

`-opt-jump-tables=default` or `/Qopt-jump-tables:default`	The compiler uses default heuristics to determine when to generate jump tables for switch statements.

Description

This option enables or disables generation of jump tables for switch statements. When the option is enabled, it may improve performance for programs with large switch statements.

Alternate Options

None

opt-loadpair, Qopt-loadpair

Enables or disables loadpair optimization.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-loadpair`

`-no-opt-loadpair`

Mac OS X:

`None`

Windows:

`/Qopt-loadpair`

`/Qopt-loadpair-`

Arguments

None

Default

`-no-opt-loadpair` or `/Qopt-loadpair-`	Loadpair optimization is disabled unless option `O3` is specified.

Description

This option enables or disables loadpair optimization.

When `-O3` is specified on IA-64 architecture, loadpair optimization is enabled by default. To disable loadpair generation, specify `-no-opt-loadpair` (Linux) or `/Qopt-loadpair-` (Windows).

Alternate Options

None

opt-mem-bandwidth, Qopt-mem-bandwidth

Enables performance tuning and heuristics that control memory bandwidth use among processors.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-mem-bandwidth`*n*

Mac OS X:

`None`

Windows:

`/Qopt-mem-bandwidth`*n*

Arguments

n Is the level of optimizing for memory bandwidth usage. Possible values are:

`0`	Enables a set of performance tuning and heuristics in compiler optimizations that is optimal for serial code.
`1`	Enables a set of performance tuning and heuristics in compiler optimizations for multithreaded code generated by the compiler.
`2`	Enables a set of performance tuning and heuristics in compiler optimizations for parallel code such as Windows Threads, pthreads, and MPI code, besides multithreaded code generated by the compiler.

Default

`-opt-mem-bandwidth0` or `/Qopt-mem-bandwidth0` For serial (non-parallel) compilation, a set of performance tuning and heuristics in compiler optimizations is enabled that is optimal for serial code.

`-opt-mem-bandwidth1` or `/Qopt-mem-bandwidth1`	If you specify compiler option `-parallel` (Linux) or `/Qparallel` (Windows), or `-openmp` (Linux) or `/Qopenmp` (Windows), a set of performance tuning and heuristics in compiler optimizations for multithreaded code generated by the compiler is enabled.

Description

This option enables performance tuning and heuristics that control memory bandwidth use among processors. It allows the compiler to be less aggressive with optimizations that might consume more bandwidth, so that the bandwidth can be well-shared among multiple processors for a parallel program.

For values of *n* greater than 0, the option tells the compiler to enable a set of performance tuning and heuristics in compiler optimizations such as prefetching, privatization, aggressive code motion, and so forth, for reducing memory bandwidth pressure and balancing memory bandwidth traffic among threads.

This option can improve performance for threaded or parallel applications on multiprocessors or multicore processors, especially when the applications are bounded by memory bandwidth.

Alternate Options

None

See Also

-
-
- `parallel`, `Qparallel`
- `openmp`, `Qopenmp`

opt-mod-versioning, Qopt-mod-versioning

Enables or disables versioning of modulo operations for certain types of operands.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-mod-versioning`

`-no-opt-mod-versioning`

Mac OS X:

`None`

Windows:

`/Qopt-mod-versioning`

`/Qopt-mod-versioning-`

Arguments

None

Default

`-no-opt-mod-versioning` Versioning of modulo operations is disabled.
or `/Qopt-mod-versioning-`

Description

This option enables or disables versioning of modulo operations for certain types of operands. It is used for optimization tuning.

Versioning of modulo operations may improve performance for x mod y when modulus y is a power of 2.

Alternate Options

None

opt-multi-version-aggressive, Qopt-multi-version-aggressive

Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-multi-version-aggressive`

`-no-opt-multi-version-aggressive`

Windows:

`/Qopt-multi-version-aggressive`

`/Qopt-multi-version-aggressive-`

Arguments

None

Default

`-no-opt-multi-version-aggressive` or `/Qopt-multi-version-aggressive-`	The compiler uses default heuristics when checking for pointer aliasing and scalar replacement.

Description

This option tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement. This option may improve performance.

Alternate Options

None

opt-prefetch, Qopt-prefetch

Enables or disables prefetch insertion optimization.

IDE Equivalent

Windows: **Optimization > Prefetch Insertion**

Linux: None

Mac OS X: **Optimization > Enable Prefetch Insertion**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-prefetch[=n]`

`-no-opt-prefetch`

Windows:

`/Qopt-prefetch[:n]`

`/Qopt-prefetch-`

Arguments

n	Is the level of detail in the report. Possible values are:	
	`0`	Disables software prefetching. This is the same as specifying `-no-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch-` (Windows).
	`1` to `4`	Enables different levels of software prefetching. If you do not specify a value for *n*, the default is `2` on IA-32 and Intel® 64 architecture; the default is `3` on IA-64 architecture. Use lower values to reduce the amount of prefetching.

Default

IA-64 architecture: `-opt-prefetch` or `/Qopt-prefetch`	On IA-64 architecture, prefetch insertion optimization is enabled.
IA-32 architecture and Intel® 64 architecture: `-no-opt-prefetch` or `/Qopt-prefetch-`	On IA-32 architecture and Intel® 64 architecture, prefetch insertion optimization is disabled.

Description

This option enables or disables prefetch insertion optimization. The goal of prefetching is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache.

On IA-64 architecture, this option is enabled by default if you specify option `O1` or higher. To disable prefetching at these optimization levels, specify `-no-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch-` (Windows).

On IA-32 architecture and Intel® 64 architecture, this option enables prefetching when higher optimization levels are specified.

Alternate Options

Linux and Mac OS X: `-prefetch` (this is a deprecated option)

Windows: `/Qprefetch` (this is a deprecated option)

opt-prefetch-initial-values, Qopt-prefetch-initial-values

Enables or disables prefetches that are issued before a loop is entered.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

```
-opt-prefetch-initial-values
-no-opt-prefetch-initial-values
```

Mac OS X:

```
None
```

Windows:

```
/Qopt-prefetch-initial-values
/Qopt-prefetch-initial-values-
```

Arguments

None

Default

`-opt-prefetch-initial-values` or `/Qopt-prefetch-initial-values` Prefetches are issued before a loop is entered.

Description

This option enables or disables prefetches that are issued before a loop is entered. These prefetches target the initial iterations of the loop.

When `-O1` or higher is specified on IA-64 architecture, prefetches are issued before a loop is entered. To disable these prefetches, specify `-no-opt-prefetch-initial-values` (Linux) or `/Qopt-prefetch-initial-values-` (Windows).

Alternate Options

None

opt-prefetch-issue-excl-hint, Qopt-prefetch-issue-excl-hint

Determines whether the compiler issues prefetches for stores with exclusive hint.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-prefetch-issue-excl-hint`

`-no-opt-prefetch-issue-excl-hint`

Mac OS X:

None

Windows:

/Qopt-prefetch-issue-excl-hint

/Qopt-prefetch-issue-excl-hint-

Arguments

None

Default

-no-opt-prefetch-issue-excl-hint or /Qopt-prefetch-issue-excl-hint-	The compiler does not issue prefetches for stores with exclusive hint.

Description

This option determines whether the compiler issues prefetches for stores with exclusive hint. If option -opt-prefetch-issue-excl-hint (Linux) or /Qopt-prefetch-issue-excl-hint (Windows) is specified, the prefetches will be issued if the compiler determines it is beneficial to do so.

When prefetches are issued for stores with exclusive-hint, the cache-line is in "exclusive-mode". This saves on cache-coherence traffic when other processors try to access the same cache-line. This feature can improve performance tuning.

Alternate Options

None

opt-prefetch-next-iteration, Qopt-prefetch-next-iteration

Enables or disables prefetches for a memory access in the next iteration of a loop.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux:

`-opt-prefetch-next-iteration`

`-no-opt-prefetch-next-iteration`

Mac OS X:

`None`

Windows:

`/Qopt-prefetch-next-iteration`

`/Qopt-prefetch-next-iteration-`

Arguments

None

Default

`-opt-prefetch-next-iteration` or `/Qopt-prefetch-next-iteration`	Prefetches are issued for a memory access in the next iteration of a loop.

Description

This option enables or disables prefetches for a memory access in the next iteration of a loop. It is typically used in a pointer-chasing loop.

When `-O1` or higher is specified on IA-64 architecture, prefetches are issued for a memory access in the next iteration of a loop. To disable these prefetches, specify `-no-opt-prefetch-next-iteration` (Linux) or `/Qopt-prefetch-next-iteration-` (Windows).

Alternate Options

None

opt-ra-region-strategy, Qopt-ra-region-strategy

Selects the method that the register allocator uses to partition each routine into regions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-ra-region-strategy[=`*keyword*`]`

Windows:

`/Qopt-ra-region-strategy[:`*keyword*`]`

Arguments

keyword	Is the method used for partitioning. Possible values are:	
	`routine`	Creates a single region for each routine.
	`block`	Partitions each routine into one region per basic block.
	`trace`	Partitions each routine into one region per trace.
	`region`	Partitions each routine into one region per loop.
	`default`	The compiler determines which method is used for partitioning.

Default

`-opt-ra-region-strategy=default` or `/Qopt-ra-region-strategy:default`	The compiler determines which method is used for partitioning. This is also the default if `keyword` is not specified.

Description

This option selects the method that the register allocator uses to partition each routine into regions.

When setting `default` is in effect, the compiler attempts to optimize the tradeoff between compile-time performance and generated code performance.

This option is only relevant when optimizations are enabled (`O1` or higher).

Alternate Options

None

See Also

-
-
- `O`

opt-report, Qopt-report

Tells the compiler to generate an optimization report to stderr.

IDE Equivalent

Windows: Diagnostics > Optimization Diagnostics Level

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report [n]`

Windows:

`/Qopt-report[:n]`

Arguments

n	Is the level of detail in the report. On Linux OS and Mac OS X systems, a space must appear before the *n*. Possible values are:
	`0` Tells the compiler to generate no optimization report.
	`1` Tells the compiler to generate a report with the minimum level of detail.
	`2` Tells the compiler to generate a report with the medium level of detail.
	`3` Tells the compiler to generate a report with the maximum level of detail.

Default

`-opt-report 2` or `/Qopt-report:2`	If you do not specify *n*, the compiler generates a report with medium detail. If you do not specify the option on the command line, the compiler does not generate an optimization report.

Description

This option tells the compiler to generate an optimization report to `stderr`.

Alternate Options

None

See Also

-
-
- `opt-report-file`, `Qopt-report-file`

Optimizing Applications: Optimizer Report Generation

opt-report-file, Qopt-report-file

Specifies the name for an optimization report.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-file=`*`file`*

Windows:

`/Qopt-report-file:`*`file`*

Arguments

`file`	Is the name for the optimization report.

Default

OFF	No optimization report is generated.

Description

This option specifies the name for an optimization report. If you use this option, you do not have to specify `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

Optimizing Applications: Optimizer Report Generation

opt-report-help, Qopt-report-help

Displays the optimizer phases available for report generation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-help`

Windows:

`/Qopt-report-help`

Arguments

None

Default

OFF	No optimization reports are generated.

Description

This option displays the optimizer phases available for report generation using `-opt-report-phase` (Linux and Mac OS X) or `/Qopt-report-phase` (Windows). No compilation is performed.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`
- `opt-report-phase`, `Qopt-report-phase`

opt-report-phase, Qopt-report-phase

Specifies an optimizer phase to use when optimization reports are generated.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-phase=`*`phase`*

Windows:

`/Qopt-report-phase:`*`phase`*

Arguments

`phase` Is the phase to generate reports for. Some of the possible values are:

`ipo`	The Interprocedural Optimizer phase
`hlo`	The High Level Optimizer phase
`hpo`	The High Performance Optimizer phase
`ilo`	The Intermediate Language Scalar Optimizer phase
`ecg`	The Code Generator phase (Windows and Linux systems using IA-64 architecture only)
`ecg_swp`	The software pipelining component of the Code Generator phase (Windows and Linux systems using IA-64 architecture only)
`pgo`	The Profile Guided Optimization phase
`all`	All optimizer phases

Default

OFF	No optimization reports are generated.

Description

This option specifies an optimizer phase to use when optimization reports are generated. To use this option, you must also specify `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows).

This option can be used multiple times on the same command line to generate reports for multiple optimizer phases.

When one of the logical names for optimizer phases is specified for phase, all reports from that optimizer phase are generated.

To find all phase possibilities, use option `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows).

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

opt-report-routine, Qopt-report-routine

Tells the compiler to generate reports on the routines containing specified text.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-opt-report-routine=`*`string`*

Windows:

`/Qopt-report-routine:`*`string`*

Arguments

`string`	Is the text (string) to look for.

Default

OFF	No optimization reports are generated.

Description

This option tells the compiler to generate reports on the routines containing specified text as part of their name.

Alternate Options

None

See Also

-
-
- `opt-report`, `Qopt-report`

opt-streaming-stores, Qopt-streaming-stores

Enables generation of streaming stores for optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-opt-streaming-stores` *keyword*

Windows:

`/Qopt-streaming-stores:`*keyword*

Arguments

keyword	Specifies whether streaming stores are generated. Possible values are:	
	`always`	Enables generation of streaming stores for optimization. The compiler optimizes under the assumption that the application is memory bound.
	`never`	Disables generation of streaming stores for optimization. Normal stores are performed.
	`auto`	Lets the compiler decide which instructions to use.

Default

`-opt-streaming-stores auto` or `/Qopt-streaming-stores:auto`	The compiler decides whether to use streaming stores or normal stores.

Description

This option enables generation of streaming stores for optimization. This method stores data with instructions that use a non-temporal buffer, which minimizes memory hierarchy pollution.

For this option to be effective, the compiler must be able to generate SSE2 (or higher) instructions. For more information, see compiler option `x` or `ax`.

This option may be useful for applications that can benefit from streaming stores.

Alternate Options

None

See Also

-
-
- ax, Qax
- x, Qx
- opt-mem-bandwidth, Qopt-mem-bandwidth, Qx

Optimizing Applications: Vectorization Support

opt-subscript-in-range, Qopt-subscript-in-range

Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-opt-subscript-in-range
```

```
-no-opt-subscript-in-range
```

Windows:

```
/Qopt-subscript-in-range
```

```
/Qopt-subscript-in-range-
```

Arguments

None

Default

`-no-opt-subscript-in-range` or `/Qopt-subscript-in-range-`	The compiler assumes overflows in the intermediate computation of subscript expressions in loops.

Description

This option determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.

If you specify `-opt-subscript-in-range` (Linux and Mac OS X) or `/Qopt-subscript-in-range` (Windows), the compiler ignores any data type conversions used and it assumes no overflows in the intermediate computation of subscript expressions. This feature can enable more loop transformations.

Alternate Options

None

Example

The following shows an example where these options can be useful. m is declared as type integer(kind=8) (64-bits) and all other variables inside the subscript are declared as type integer(kind=4) (32-bits):

```
A[ i  + j  + ( n  + k) * m ]
```

Qoption

Passes options to a specified tool.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Qoption,`*`string`*`,`*`options`*

Windows:

`/Qoption,string,options`

Arguments

string	Is the name of the tool.
options	Are one or more comma-separated, valid options for the designated tool.

Default

OFF	No options are passed to tools.

Description

This option passes options to a specified tool.

If an argument contains a space or tab character, you must enclose the entire argument in quotation marks (" "). You must separate multiple arguments with commas.

string can be any of the following:

- fpp (or cpp) - Indicates the Intel Fortran preprocessor.
- asm - Indicates the assembler.
- link - Indicates the linker.
- prof - Indicates the profiler.
- On Windows systems, the following is also available:
  - masm - Indicates the Microsoft assembler.
- On Linux and Mac OS X systems, the following are also available:
  - as - Indicates the assembler.
  - gas - Indicates the GNU assembler.
  - ld - Indicates the loader.
  - gld - Indicates the GNU loader.
  - lib - Indicates an additional library.
  - crt - Indicates the crt%.o files linked into executables to contain the place to start execution.

Alternate Options

None

Example

On Linux and Mac OS X systems:

The following example directs the linker to link with an alternative library:

```
ifort -Qoption,link,-L.,-Lmylib prog1.f
```

The following example passes a compiler option to the assembler to generate a listing file:

```
ifort -Qoption,as,"-as=myprogram.lst" -use-asm myprogram.f90
```

On Windows systems:

The following example directs the linker to create a memory map when the compiler produces the executable file from the source being compiled:

```
ifort /Qoption,link,/map:prog1.map prog1.f
```

The following example passes a compiler option to the assembler:

```
ifort /Quse_asm /Qoption,masm,"/WX" myprogram.f90
```

See Also

-
- Qlocation

qp

See p.

pad, Qpad

Enables the changing of the variable and array memory layout.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-pad`

`-nopad`

Windows:

`/Qpad`

`/Qpad-`

Arguments

None

Default

`-nopad` or `/Qpad-` Variable and array memory layout is performed by default methods.

Description

This option enables the changing of the variable and array memory layout.

This option is effectively not different from the `align` option when applied to structures and derived types. However, the scope of `pad` is greater because it applies also to common blocks, derived types, sequence types, and structures.

Alternate Options

None

See Also

-
-
- `align`

pad-source, Qpad-source

Specifies padding for fixed-form source records.

IDE Equivalent

Windows: **Language > Pad Fixed Form Source Lines**

Linux: None

Mac OS X: **Language > Pad Fixed Form Source Lines**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-pad-source`

`-nopad-source`

Windows:

`/pad-source`

`/nopad-source`

`/Qpad-source`

`/Qpad-source-`

Arguments

None

Default

`-nopad-source` or `/Qpad-source-` Fixed-form source records are not padded.

Description

This option specifies padding for fixed-form source records. It tells the compiler that fixed-form source lines shorter than the statement field width are to be padded with spaces to the end of the statement field. This affects the interpretation of character and Hollerith literals that are continued across source records.

The default value setting causes a warning message to be displayed if a character or Hollerith literal that ends before the statement field ends is continued onto the next source record. To suppress this warning message, specify option `-warn nousage` (Linux and Mac OS X) or `/warn:nousage` (Windows).

Specifying `pad-source` or `/Qpad-source` can prevent warning messages associated with option `-warn usage` (Linux and Mac OS X) or `/warn:usage` (Windows).

Alternate Options

None

See Also

-
-
- `warn`

Qpar-adjust-stack

Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Qpar-adjust-stack:`*n*

Arguments

n	Is the stack size (in bytes) for the fiber-based main thread. It must be a number equal to or greater than zero.

Default

`/Qpar-adjust-stack:0`	No adjustment is made to the main thread stack size.

Description

This option tells the compiler to generate code to adjust the stack size for a fiber-based main thread. This can reduce the stack size of threads.

For this option to be effective, you must also specify option `/Qparallel`.

Alternate Options

None

See Also

-
- `parallel`, `Qparallel`

par-affinity, Qpar-affinity

Specifies thread affinity.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-par-affinity=[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

Mac OS X:

`None`

Windows:

`/Qpar-affinity:[`*`modifier`*`,...]`*`type`*`[,`*`permute`*`][,`*`offset`*`]`

Arguments

`modifier`	Is one of the following values: `granularity={fine|thread|core}`, `[no]respect`, `[no]verbose`, `[no]warnings`, `proclist=proc_list`. The

	default is `granularity=core, respect, and noverbose`. For information on value `proclist`, see Thread Affinity Interface in *Optimizing Applications*.
type	Indicates the thread affinity. This argument is required and must be one of the following values: `compact`, `disabled`, `explicit`, `none`, `scatter`, `logical`, `physical`. The default is `none`. Values `logical` and `physical` are deprecated. Use `compact` and `scatter`, respectively, with no *permute* value.
permute	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.
offset	Is a positive integer. You cannot use this argument with *type* setting `explicit`, `none`, or `disabled`. The default is 0.

Default

OFF	The thread affinity is determined by the run-time environment.

Description

This option specifies thread affinity, which binds threads to physical processing units. It has the same effect as environment variable KMP_AFFINITY.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- Linux* OS: You have specified option `-parallel` or `-openmp` (or both).

  Windows* OS: You have specified option `/Qparallel` or `/Qopenmp` (or both).
- You are compiling the main program.

Alternate Options

None

See Also

-
-

- Thread Affinity Interface

par-num-threads, Qpar-num-threads

Specifies the number of threads to use in a parallel region.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-num-threads=`*n*

Windows:

`/Qpar-num-threads:`*n*

Arguments

n	Is the number of threads to use. It must be a positive integer.

Default

OFF	The number of threads to use is determined by the run-time environment.

Description

This option specifies the number of threads to use in a parallel region. It has the same effect as environment variable OMP_NUM_THREADS.

This option overrides the environment variable when both are specified.

This option only has an effect if the following is true:

- Linux* OS and Mac OS* X: You have specified option `-parallel` or `-openmp` (or both).

  Windows* OS: You have specified option `/Qparallel` or `/Qopenmp` (or both).

- You are compiling the main program.

Alternate Options

None

par-report, Qpar-report

Controls the diagnostic information reported by the auto-parallelizer.

IDE Equivalent

Windows: **Compilation Diagnostics > Auto-Parallelizer Diagnostic Level**

Linux: None

Mac OS X: **Diagnostics > Auto-Parallelizer Report**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-report[n]`

Windows:

`/Qpar-report[n]`

Arguments

n	Is a value denoting which diagnostic messages to report. Possible values are:
`0`	Tells the auto-parallelizer to report no diagnostic information.
`1`	Tells the auto-parallelizer to report diagnostic messages for loops successfully auto-parallelized. The compiler also issues a "LOOP AUTO-PARALLELIZED" message for parallel loops.

2	Tells the auto-parallelizer to report diagnostic messages for loops successfully and unsuccessfully auto-parallelized.
3	Tells the auto-parallelizer to report the same diagnostic messages specified by 2 plus additional information about any proven or assumed dependencies inhibiting auto-parallelization (reasons for not parallelizing).

Default

`-par-report1` or `/Qpar-report1`	If you do not specify *n*, the compiler displays diagnostic messages for loops successfully auto-parallelized. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the diagnostic information reported by the auto-parallelizer (parallel optimizer). To use this option, you must also specify `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows).

If this option is specified on the command line, the report is sent to `stdout`.

On Windows systems, if this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

par-runtime-control, Qpar-runtime-control

Generates code to perform run-time checks for loops that have symbolic loop bounds.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-runtime-control`

`-no-par-runtime-control`

Windows:

`/Qpar-runtime-control`

`/Qpar-runtime-control-`

Arguments

None

Default

`-no-par-runtime-control` The compiler uses default heuristics when checking loops.
or `/Qpar-runtime-control-`

Description

This option generates code to perform run-time checks for loops that have symbolic loop bounds.

If the granularity of a loop is greater than the parallelization threshold, the loop will be executed in parallel.

If you do not specify this option, the compiler may not parallelize loops with symbolic loop bounds if the compile-time granularity estimation of a loop can not ensure it is beneficial to parallelize the loop.

Alternate Options

None

par-schedule, Qpar-schedule

Lets you specify a scheduling algorithm or a tuning method for loop iterations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-schedule-`*`keyword`*`[=`*`n`*`]`

Windows:

`/Qpar-schedule-`*`keyword`*`[[:]`*`n`*`]`

Arguments

keyword	Specifies the scheduling algorithm or tuning method. Possible values are:	
	`auto`	Lets the compiler or run-time system determine the scheduling algorithm.
	`static`	Divides iterations into contiguous pieces.
	`static-balanced`	Divides iterations into even-sized chunks.
	`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
	`dynamic`	Gets a set of iterations dynamically.
	`guided`	Specifies a minimum number of iterations.
	`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
	`runtime`	Defers the scheduling decision until run time.
n	Is the size of the chunk or the number of iterations for each chunk. This setting can only be specified for static, dynamic, and guided. For more information, see the descriptions of each keyword below.	

Default

static-balanced	Iterations are divided into even-sized chunks and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

Description

This option lets you specify a scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.

This option affects performance tuning and can provide better performance during auto-parallelization.

Option	Description
`-par-schedule-auto` or `/Qpar-schedule-auto`	Lets the compiler or run-time system determine the scheduling algorithm. Any possible mapping may occur for iterations to threads in the team.
`-par-schedule-static` or `/Qpar-schedule-static`	Divides iterations into contiguous pieces (chunks) of size *n*. The chunks are assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. If no *n* is specified, the iteration space is divided into chunks that are approximately equal in size, and each thread is assigned at most one chunk.
`-par-schedule-static-balanced` or `/Qpar-schedule-static-balanced`	Divides iterations into even-sized chunks. The chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
`-par-schedule-static-steal` or `/Qpar-schedule-static-steal`	Divides iterations into even-sized chunks, but when a thread completes its chunk, it can steal parts of chunks assigned to neighboring threads.

Option	Description
	Each thread keeps track of L and U, which represent the lower and upper bounds of its chunks respectively. Iterations are executed starting from the lower bound, and simultaneously, L is updated to represent the new lower bound.
`-par-schedule-dynamic` or `/Qpar-schedule-dynamic`	Can be used to get a set of iterations dynamically. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations. Each chunk contains *n* iterations, except for the last chunk to be assigned, which may have fewer iterations. If no *n* is specified, the default is 1.
`-par-schedule-guided` or `/Qpar-schedule-guided`	Can be used to specify a minimum number of iterations. Assigns iterations to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk of size 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For an *n* with value *k* (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). If no *n* is specified, the default is 1.

Option	Description
`-par-schedule-guided-analytical` or `/Qpar-schedule-guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution. The method depends on run-time implementation. Loop bounds are calculated with faster synchronization and chunks are dynamically dispatched at run time by threads in the team.
`-par-schedule-runtime` or `/Qpar-schedule-runtime`	Defers the scheduling decision until run time. The scheduling algorithm and chunk size are then taken from the setting of environment variable OMP_SCHEDULE.

Alternate Options

None

par-threshold, Qpar-threshold

Sets a threshold for the auto-parallelization of loops.

IDE Equivalent

Windows: **Optimization > Threshold For Auto-Parallelization**

Linux: None

Mac OS X: **Optimization > Threshold For Auto-Parallelization**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-par-threshold[`*n*`]`

Windows:

`/Qpar-threshold[[:]`*n*`]`

Arguments

n	Is an integer whose value is the threshold for the auto-parallelization of loops. Possible values are 0 through 100. If *n* is 0, loops get auto-parallelized always, regardless of computation work volume. If *n* is 100, loops get auto-parallelized when performance gains are predicted based on the compiler analysis data. Loops get auto-parallelized only if profitable parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to parallelize only if there is a 50% probability of the code speeding up if executed in parallel.

Default

`-par-threshold100` or `/Qpar-threshold100`	Loops get auto-parallelized only if profitable parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. To use this option, you must also specify `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows).

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Alternate Options

None

parallel, Qparallel

Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

IDE Equivalent

Windows: **Optimization > Parallelization**

Linux: None

Mac OS X: **Optimization > Parallelization**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-parallel`

Windows:

`/Qparallel`

Arguments

None

Default

OFF	Multithreaded code is not generated for loops that can be safely executed in parallel.

Description

This option tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

To use this option, you must also specify option `O2` or `O3`.

NOTE. On Mac OS X systems, when you enable automatic parallelization, you must also set the `DYLD_LIBRARY_PATH` environment variable within Xcode or an error will be displayed.

Alternate Options

None

See Also

-
-
- `par-report`, `Qpar-report`
- `par-affinity`, `Qpar-affinity`
- `par-num-threads`, `Qpar-num-threads`
- `par-runtime-control`, `Qpar-runtime-control`
- `par-schedule`, `Qpar-schedule`
- `O`

pc, Qpc

Enables control of floating-point significand precision.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-pc`*n*

Windows:

`/Qpc`*n*

Arguments

n	Is the floating-point significand precision. Possible values are:
	`32` Rounds the significand to 24 bits (single precision).
	`64` Rounds the significand to 53 bits (double precision).
	`80` Rounds the significand to 64 bits (extended precision).

Default

`-pc80` or `/Qpc64`	On Linux* and Mac OS* X systems, the floating-point significand is rounded to 64 bits. On Windows* systems, the floating-point significand is rounded to 53 bits.

Description

This option enables control of floating-point significand precision.

Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the this option.

Note that a change of the default precision control or rounding mode, for example, by using the `-pc32` (Linux and Mac OS X) or `/Qpc32` (Windows) option or by user intervention, may affect the results returned by some of the mathematical functions.

Alternate Options

None

See Also

-
-

Floating-point Operations: Floating-point Options Quick Reference

mp1, Qprec

Improves floating-point precision and consistency.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-mp1`

Windows:

`/Qprec`

Arguments

None

Default

OFF	The compiler provides good accuracy and run-time performance at the expense of less consistent floating-point results.

Description

This option improves floating-point consistency. It ensures the out-of-range check of operands of transcendental functions and improves the accuracy of floating-point compares.

This option prevents the compiler from performing optimizations that change NaN comparison semantics and causes all values to be truncated to declared precision before they are used in comparisons. It also causes the compiler to use library routines that give better precision results compared to the X87 transcendental instructions.

This option disables fewer optimizations and has less impact on performance than option `flt-consistency` or `mp`.

Alternate Options

None

See Also

-
-
- fltconsistency
- mp

prec-div, Qprec-div

Improves precision of floating-point divides.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

-prec-div

-no-prec-div

Windows:

/Qprec-div

/Qprec-div-

Arguments

None

Default

-prec-div or/Qprec-div	The compiler uses this method for floating-point divides.

Description

This option improves precision of floating-point divides. It has a slight impact on speed.

With some optimizations, such as `-xSSE2` (Linux) or `/QxSSE2` (Windows), the compiler may change floating-point division computations into multiplication by the reciprocal of the denominator. For example, A/B is computed as A * (1/B) to improve the speed of the computation.

However, sometimes the value produced by this transformation is not as accurate as full IEEE division. When it is important to have fully precise IEEE division, use this option to disable the floating-point division-to-multiplication optimization. The result is more accurate, with some loss of performance.

If you specify `-no-prec-div` (Linux and Mac OS X) or `/Qprec-div-` (Windows), it enables optimizations that give slightly less precise results than full IEEE division.

Alternate Options

None

See Also

-
-

Floating-point Operations: Floating-point Options Quick Reference

prec-sqrt, Qprec-sqrt

Improves precision of square root implementations.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-prec-sqrt
```

```
-no-prec-sqrt
```

Windows:

```
/Qprec-sqrt
```

/Qprec-sqrt-

Arguments

None

Default

-no-prec-sqrt or /Qprec-sqrt-	The compiler uses a faster but less precise implementation of square root. However, the default is -prec-sqrt or /Qprec-sqrt if any of the following options are specified: /Od, /Op, or /Qprec on Windows systems; -O0, -mp (or -fltconsistency), or -mp1 on Linux and Mac OS X systems.

Description

This option improves precision of square root implementations. It has a slight impact on speed.

This option inhibits any optimizations that can adversely affect the precision of a square root computation. The result is fully precise square root implementations, with some loss of performance.

Alternate Options

None

prof-data-order, Qprof-data-order

Enables or disables data ordering if profiling information is enabled.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

-prof-data-order

`-no-prof-data-order`

Mac OS X:

`None`

Windows:

`/Qprof-data-order`

`/Qprof-data-order-`

Arguments

None

Default

`-no-prof-data-order` or `/Qprof-data-order-`	Data ordering is disabled.

Description

This option enables or disables data ordering if profiling information is enabled. It controls the use of profiling information to order static program data items.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows).
- For feedback compilation, you must specify `-prof-use` (Linux) or `/Qprof-use` (Windows). You must not use multi-file optimization by specifying options such as option `-ipo` (Linux) or `/Qipo` (Windows), or option `-ipo-c` (Linux) or `/Qipo-c` (Windows).

Alternate Options

None

See Also

-
-
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-func-order, Qprof-func-order

prof-dir, Qprof-dir

Specifies a directory for profiling information output files.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-dir` *dir*

Windows:

`/Qprof-dir` *dir*

Arguments

dir	Is the name of the directory.

Default

OFF	Profiling output files are placed in the directory where the program is compiled.

Description

This option specifies a directory for profiling information output files (*.dyn and *.dpi). The specified directory must already exist.

You should specify this option using the same directory name for both instrumentation and feedback compilations. If you move the .dyn files, you need to specify the new path.

Alternate Options

None

See Also

•

-

Floating-point Operations:

Profile-guided Optimization (PGO) Quick Reference

Coding Guidelines for Intel(R) Architectures

prof-file, Qprof-file

Specifies an alternate file name for the profiling summary files.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-file` *`file`*

Windows:

`/Qprof-file` *`file`*

Arguments

`file`	Is the name of the profiling summary file.

Default

OFF	The profiling summary files have the file name pgopti.*

Description

This option specifies an alternate file name for the profiling summary files. The *`file`* is used as the base name for files created by different profiling passes.

If you add this option to profmerge, the .dpi file will be named *`file`*.dpi instead of pgopti.dpi.

If you specify `-prof-genx` (Linux and Mac OS X) or `/Qprof-genx` (Windows) with this option, the .spi and .spl files will be named *`file`*.spi and *`file`*.spl instead of pgopti.spi and pgopti.spl.

If you specify `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) with this option, the .dpi file will be named *`file`*.dpi instead of pgopti.dpi.

Alternate Options

None

See Also

-
-
- `prof-gen, Qprof-gen`
- `prof-use, Qprof-use`

Optimizing Applications:

Profile-guided Optimizations Overview

Coding Guidelines for Intel(R) Architectures

Profile an Application

prof-func-order, Qprof-func-order

Enables or disables function ordering if profiling information is enabled.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-prof-func-order`

`-no-prof-func-order`

Mac OS X:

`None`

Windows:

`/Qprof-func-order`

`/Qprof-func-order-`

Arguments

None

Default

`-no-prof-func-order` or `/Qprof-func-order-`	Function ordering is disabled.

Description

This option enables or disables function ordering if profiling information is enabled.

For this option to be effective, you must do the following:

- For instrumentation compilation, you must specify `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows).
- For feedback compilation, you must specify `-prof-use` (Linux) or `/Qprof-use` (Windows). You must not use multi-file optimization by specifying options such as option `-ipo` (Linux) or `/Qipo` (Windows), or option `-ipo-c` (Linux) or `/Qipo-c` (Windows).

If you enable profiling information by specifying option `-prof-use` (Linux) or `/Qprof-use` (Windows), `-prof-func-groups` (Linux) and `/Qprof-func-groups` (Windows) are set and function grouping is enabled. However, if you explicitly enable `-prof-func-order` (Linux) or `/Qprof-func-order` (Windows), function ordering is performed instead of function grouping.

On Linux* systems, this option is only available for Linux linker 2.15.94.0.1, or later.

To set the hotness threshold for function grouping and function ordering, use option `-prof-hotness-threshold` (Linux) or `/Qprof-hotness-threshold` (Windows).

Alternate Options

None

The following example shows how to use this option on a Windows system:

```
ifort /Qprof-gen:globdata file1.f90 file2.f90 /exe:instrumented.exe
    ./instrumented.exe
ifort /Qprof-use /Qprof-func-order file1.f90 file2.f90 /exe:feedback.exe
```

The following example shows how to use this option on a Linux system:

```
ifort -prof-gen:globdata file1.f90 file2.f90 -o instrumented
    ./instrumented.exe
ifort -prof-use -prof-func-order file1.f90 file2.f90 -o feedback
```

See Also

-
-
- prof-hotness-threshold, Qprof-hotness-threshold
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-data-order, Qprof-data-order
- prof-func-groups

prof-gen, Qprof-gen

Produces an instrumented object file that can be used in profile-guided optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-gen[=keyword]`

`-no-prof-gen`

Windows:

`/Qprof-gen[:keyword]`

`/Qprof-gen-`

Arguments

keyword	Specifies details for the instrumented file. Possible values are:	
	`default`	Produces an instrumented object file. This is the same as specifying `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) with no keyword.
	`srcpos`	Produces an instrumented object file that includes extra source position information. This option is the same as option `-prof-genx` (Linux* and Mac OS* X) or `/Qprof-genx` (Windows*), which are deprecated.
	`globdata`	Produces an instrumented object file that includes information for global data layout.

Default

`-no-prof-gen` or `/Qprof-` Profile generation is disabled.
`gen-`

Description

This option produces an instrumented object file that can be used in profile-guided optimization. It gets the execution count of each basic block.

If you specify keyword `srcpos` or `globdata`, a static profile information file (.spi) is created. These settings may increase the time needed to do a parallel build using -prof-gen, because of contention writing the .spi file.

These options are used in phase 1 of the Profile Guided Optimizer (PGO) to instruct the compiler to produce instrumented code in your object files in preparation for instrumented execution.

Alternate Options

None

See Also

•
•

Optimizing Applications:

Basic PGO Options

Example of Profile-Guided Optimization

prof-genx, Qprof-genx

This is a deprecated option. See prof-gen keyword `srcpos`*.*

prof-hotness-threshold, Qprof-hotness-threshold

Lets you set the hotness threshold for function grouping and function ordering.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-prof-hotness-threshold=`*n*

Mac OS X:

`None`

Windows:

`/Qprof-hotness-threshold:`*n*

Arguments

n	Is the hotness threshold. *n* is a percentage having a value between 0 and 100 inclusive. If you specify 0, there will be no hotness threshold setting in effect for function grouping and function ordering.

Default

OFF	The compiler's default hotness threshold setting of 10 percent is in effect for function grouping and function ordering.

Description

This option lets you set the hotness threshold for function grouping and function ordering.

The "hotness threshold" is the percentage of functions in the application that should be placed in the application's hot region. The hot region is the most frequently executed part of the application. By grouping these functions together into one hot region, they have a greater probability of remaining resident in the instruction cache. This can enhance the application's performance.

For this option to take effect, you must specify option `-prof-use` (Linux) or `/Qprof-use` (Windows) and one of the following:

- On Linux systems: `-prof-func-groups` or `-prof-func-order`
- On Windows systems: `/Qprof-func-order`

Alternate Options

None

See Also

-
-
- prof-use, Qprof-use
- prof-func-groups
- prof-func-order, Qprof-func-order

prof-src-dir, Qprof-src-dir

Determines whether directory information of the source file under compilation is considered when looking up profile data records.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-src-dir`

`-no-prof-src-dir`

Windows:

`/Qprof-src-dir`

`/Qprof-src-dir-`

Arguments

None

Default

`-prof-src-dir` or `/Qprof-src-dir`	Directory information is used when looking up profile data records in the .dpi file.

Description

This option determines whether directory information of the source file under compilation is considered when looking up profile data records in the .dpi file. To use this option, you must also specify option `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

If the option is enabled, directory information is considered when looking up the profile data records within the .dpi file. You can specify directory information by using one of the following options:

- Linux and Mac OS X: `-prof-src-root` or `-prof-src-root-cwd`
- Windows: `/Qprof-src-root` or `/Qprof-src-root-cwd`

If the option is disabled, directory information is ignored and only the name of the file is used to find the profile data record.

Note that options `-prof-src-dir` (Linux and Mac OS X) and `/Qprof-src-dir` (Windows) control how the names of the user's source files get represented within the .dyn or .dpi files. Options `-prof-dir` (Linux and Mac OS X) and `/Qprof-dir` (Windows) specify the location of the .dyn or the .dpi files.

Alternate Options

None

See Also

-
-
- `prof-use, Qprof-use`
- `prof-src-root, Qprof-src-root`
- `prof-src-root-cwd, Qprof-src-root-cwd`

prof-src-root, Qprof-src-root

Lets you use relative directory paths when looking up profile data and specifies a directory as the base.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-src-root=`*`dir`*

Windows:

`/Qprof-src-root:`*`dir`*

Arguments

`dir`	Is the base for the relative paths.

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It lets you specify a directory as the base. The paths are relative to a base directory specified during the `-prof-gen` (Linux and Mac OS X) or `/Qprof-gen` (Windows) compilation phase.

This option is available during the following phases of compilation:

- Linux and Mac OS X: `-prof-gen` and `-prof-use` phases
- Windows: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `-prof-gen` or `/Qprof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `-prof-use` or `/Qprof-use` phase, it specifies a root directory that replaces the root directory specified at the `-prof-gen` or `/Qprof-gen` phase for forming the lookup keys.

To be effective, this option or option `-prof-src-root-cwd` (Linux and Mac OS X) or `/Qprof-src-root-cwd` (Windows) must be specified during the `-prof-gen` or `/Qprof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

Alternate Options

None

Consider the initial `-prof-gen` compilation of the source file c:\user1\feature_foo\myproject\common\glob.f90:

```
ifort -prof-gen -prof-src-root=c:\user1\feature_foo\myproject -c common\glob.f90
```

For the `-prof-use` phase, the file glob.f90 could be moved into the directory c:\user2\feature_bar\myproject\common\glob.f90 and profile information would be found from the .dpi when using the following:

```
ifort -prof-use -prof-src-root=c:\user2\feature_bar\myproject -c common\glob.f90
```

If you do not use option `-prof-src-root` during the `-prof-gen` phase, by default, the `-prof-use` compilation can only find the profile data if the file is compiled in the c:\user1\feature_foo\my_project\common directory.

See Also

-
-
- prof-gen, Qprof-gen
- prof-use, Qprof-use
- prof-src-dir, Qprof-src-dir
- prof-src-root-cwd, Qprof-src-root-cwd

prof-src-root-cwd, Qprof-src-root-cwd

Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-prof-src-root-cwd
```

Windows:

```
/Qprof-src-root-cwd
```

Arguments

None

Default

OFF	The setting of relevant options determines the path used when looking up profile data records.

Description

This option lets you use relative directory paths when looking up profile data in .dpi files. It specifies the current working directory as the base. To use this option, you must also specify option `-prof-use` (Linux and Mac OS) or `/Qprof-use` (Windows).

This option is available during the following phases of compilation:

- Linux and Mac OS X: `-prof-gen` and `-prof-use` phases
- Windows: `/Qprof-gen` and `/Qprof-use` phases

When this option is specified during the `-prof-gen` or `/Qprof-gen` phase, it stores information into the .dyn or .dpi file. Then, when .dyn files are merged together or the .dpi file is loaded, only the directory information below the root directory is used for forming the lookup key.

When this option is specified during the `-prof-use` or `/Qprof-use` phase, it specifies a root directory that replaces the root directory specified at the `-prof-gen` or `/Qprof-gen` phase for forming the lookup keys.

To be effective, this option or option `-prof-src-root` (Linux and Mac OS X) or `/Qprof-src-root` (Windows) must be specified during the `-prof-gen` or `/Qprof-gen` phase. In addition, if one of these options is not specified, absolute paths are used in the .dpi file.

Alternate Options

None

See Also

-
-
- `prof-gen, Qprof-gen`
- `prof-use, Qprof-use`
- `prof-src-dir, Qprof-src-dir`
- `prof-src-root, Qprof-src-root`

prof-use, Qprof-use

Enables the use of profiling information during optimization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-prof-use[=arg]`

`-no-prof-use`

Windows:

`/Qprof-use[:arg]`

`/Qprof-use-`

Arguments

arg Specifies additional instructions. Possible values are:

`weighted` Tells the profmerge utility to apply a weighting to the .dyn file values when creating the .dpi file to normalize the data counts when the training runs have differentexecution durations. This argument only has an effect when the compiler invokes the profmerge utility to create the .dpi file. This argument does not have an effect if the .dpi file was previously created without weighting.

`[no]merge` Enables or disables automatic invocation of the profmerge utility. The default is `merge`. Note that you cannot specify both `weighted` and `nomerge`. If you try to specify both values, a warning will be displayed and `nomerge` takes precedence.

`default` Enables the use of profiling information during optimization. The profmerge utility is invoked by default. This value is the same as specifying `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) with no argument.

Default

`-no-prof-use` or `/Qprof-use-` Profiling information is not used during optimization.

Description

This option enables the use of profiling information (including function splitting and function grouping) during optimization. It enables option `-fnsplit` (Linux) or `/Qfnsplit` (Windows).

This option instructs the compiler to produce a profile-optimized executable and it merges available profiling output files into a pgopti.dpi file.

Note that there is no way to turn off function grouping if you enable it using this option.

To set the hotness threshold for function grouping and function ordering, use option `-prof-hotness-threshold` (Linux) or `/Qprof-hotness-threshold` (Windows).

Alternate Options

None

See Also

-
-
- `prof-hotness-threshold, Qprof-hotness-threshold`

Optimizing Applications:

Basic PGO Options

Example of Profile-Guided Optimization

rcd, Qrcd

Enables fast float-to-integer conversions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-rcd`

Windows:

`/Qrcd`

Arguments

None

Default

OFF	Floating-point values are truncated when a conversion to an integer is involved. On Windows, this is the same as specifying `/QIfist-`.

Description

This option enables fast float-to-integer conversions. It can improve the performance of code that requires floating-point-to-integer conversions.

The system default floating-point rounding mode is round-to-nearest. However, the Fortran language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point-to-integer conversion and change it back afterwards.

This option disables the change to truncation of the rounding mode for all floating-point calculations, including floating point-to-integer conversions. This option can improve performance, but floating-point conversions to integer will not conform to Fortran semantics.

Alternate Options

Linux and Mac OS X: None

Windows: `/QIfist`

rct, Qrct

Sets the internal FPU rounding control to Truncate.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-rct`

Windows:

`/Qrct`

Arguments

None

Default

OFF	The compiler uses the default setting for the FPU rounding control.

Description

This option sets the internal FPU rounding control to Truncate.

Alternate Options

Linux and Mac OS X: None

Windows: `/rounding-mode:chopped`

safe-cray-ptr, Qsafe-cray-ptr

Tells the compiler that Cray pointers do not alias other variables.*

IDE Equivalent

Windows: **Data > Assume Cray Pointers Do Not Share Memory Locations**

Linux: None

Mac OS X: **Data > Assume Cray Pointers Do Not Share Memory Locations**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-safe-cray-ptr`

Windows:

`/Qsafe-cray-ptr`

Arguments

None

Default

OFF	The compiler assumes that Cray pointers alias other variables.

Description

This option tells the compiler that Cray pointers do not alias (that is, do not specify sharing memory with) other variables.

Alternate Options

None

Example

Consider the following:

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

By default, the compiler assumes that b and a are aliased. To prevent such an assumption, specify the `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) option, and the compiler will treat b(i) and a(i) as independent of each other.

However, if the variables are intended to be aliased with Cray pointers, using the option produces incorrect results. In the following example, you should not use the option:

```
pointer (pb, b)
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

save, Qsave

Causes variables to be placed in static memory.

IDE Equivalent

Windows: **Data > Local Variable Storage**

Linux: None

Mac OS X: **Data > Local Variable Storage**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-save
```

Windows:

```
/Qsave
```

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL are allocated to the run-time stack. Note that if option `recursive`, `-openmp` (Linux and Mac OS X), or `/Qopenmp` (Windows) is specified, the default is `-automatic` (Linux) or `/Qauto` (Windows).

Description

This option saves all variables in static allocation except local variables within a recursive routine and variables declared as AUTOMATIC.

If you want all local, non-SAVEd variables to be allocated to the run-time stack, specify option `automatic`.

Alternate Options

Linux and Mac OS X: `-noautomatic`, `-noauto`

Windows: `/noautomatic`, `/noauto`, `/4Na`

See Also

-
-
- `automatic`
- `auto_scalar`

save-temps, Qsave-temps

Tells the compiler to save intermediate files created during compilation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-save-temps`

`-no-save-temps`

Windows:

`/Qsave-temps`

`/Qsave-temps-`

Arguments

None

Default

Linux and Mac OS X: `-no-save-temps` Windows: .obj files are saved	On Linux and Mac OS X systems, the compiler deletes intermediate files after compilation is completed. On Windows systems, the compiler saves only intermediate object files after compilation is completed.

Description

This option tells the compiler to save intermediate files created during compilation. The names of the files saved are based on the name of the source file; the files are saved in the current working directory.

If `-save-temps` or `/Qsave-temps` is specified, the following occurs:

- The object .o file (Linux and Mac OS X) or .obj file (Windows) is saved.
- The assembler .s file (Linux and Mac OS X) or .asm file (Windows) is saved if you specified -use-asm (Linux or Mac OS X) or /Quse-asm (Windows).
- The .i or .i90 file is saved if the fpp preprocessor is invoked.

If `-no-save-temps` is specified on Linux or Mac OS X systems, the following occurs:

- The .o file is put into `/tmp` and deleted after calling `ld`.
- The preprocessed file is not saved after it has been used by the compiler.

If `/Qsave-temps-` is specified on Windows systems, the following occurs:

- The .obj file is not saved after the linker step.
- The preprocessed file is not saved after it has been used by the compiler.

NOTE. This option only saves intermediate files that are normally created during compilation.

Alternate Options

None

Example

If you compile program `my_foo.F` on a Linux or Mac OS X system and you specify option `-save-temps` and option `-use-asm`, the compilation will produce `files my_foo.o`, `my_foo.s`, and `my_foo.i`.

If you compile program `my_foo.fpp` on a Windows system and you specif option `/Qsave-temps` and option `/Quse-asm`, the compilation will produce `files my_foo.obj`, `my_foo.asm`, and `my_foo.i`.

scalar-rep, Qscalar-rep

Enables scalar replacement performed during loop transformation.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux and Mac OS X:

`-scalar-rep`

`-no-scalar-rep`

Windows:

`/Qscalar-rep`

/Qscalar-rep-

Arguments

None

Default

-no-scalar-rep or/Qscalar-rep-	Scalar replacement is not performed during loop transformation.

Description

This option enables scalar replacement performed during loop transformation. To use this option, you must also specify O3.

Alternate Options

None

See Also

-
-
- O

Qsfalign

Specifies stack alignment for functions.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux and Mac OS X:

None

Windows:

/Qsfalign[*n*]

Arguments

n	Is the byte size of aligned variables. Possible values are:
8	Specifies that alignment should occur for functions with 8-byte aligned variables. At this setting the compiler aligns the stack to 16 bytes if there is any 16-byte or 8-byte data on the stack. For 8-byte data, the compiler only aligns the stack if the alignment will produce a performance advantage.
16	Specifies that alignment should occur for functions with 16-byte aligned variables. At this setting, the compiler only aligns the stack for 16-byte data. No attempt is made to align for 8-byte data.

Default

/Qsfalign8	Alignment occurs for functions with 8-byte aligned variables.

Description

This option specifies stack alignment for functions. It lets you disable the normal optimization that aligns a stack for 8-byte data.

If you do not specify *n*, stack alignment occurs for all functions. If you specify /Qsfalign-, no stack alignment occurs for any function.

Alternate Options

None

sox, Qsox

Tells the compiler to save the compilation options and version number in the Linux OS executable or the Windows* OS object file.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-sox`

`-no-sox`

Mac OS X:

`None`

Windows:

`/Qsox`

`/Qsox-`

Arguments

None

Default

`-no-sox` or `/Qsox-`	The compiler does not save the compiler options and version number in the executable.

Description

Tells the compiler to save the compilation options and version number in the Linux* OS executable or the Windows* OS object file.

On Linux systems, the size of the executable on disk is increased slightly by the inclusion of these infotmation strings.

This option forces the compiler to embed in each object file or assembly output a string that contains information about the compiler version and compilation options for each source file that has been compiled.

On Windows systems, the information stays in the object file. On Linux systems, when you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

If `-no-sox` or `/Qsox-` is specified, this extra information is not put into the object or assembly output generated by the compiler.

Alternate Options

None

tcheck, Qtcheck

Enables analysis of threaded applications.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tcheck`

Mac OS X:

`None`

Windows:

`/Qtcheck`

Arguments

None

Default

OFF	Threaded applications are not instrumented by the compiler for analysis by Intel® Thread Checker.

Description

This option enables analysis of threaded applications.

To use this option, you must have Intel® Thread Checker installed, which is one of the Intel® Threading Analysis Tools. If you do not have this tool installed, the compilation will fail. Remove the `-tcheck` (Linux) or `/Qtcheck` (Windows) option from the command line and recompile.

For more information about Intel® Thread Checker (including an evaluation copy), open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

tcollect, Qtcollect

Inserts instrumentation probes calling the Intel® Trace Collector API.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tcollect[`*`lib`*`]`

Mac OS X:

`None`

Windows:

`/Qtcollect[:`*`lib`*`]`

Arguments

`lib`	Is one of the Intel® Trace Collector libraries; for example, VT, VTcs, VTmc, or VTfs. If you do not specify *`lib`*, the default library is VT.

Default

OFF	Instrumentation probes are not inserted into compiled applications.

Description

This option inserts instrumentation probes calling the Intel® Trace Collector API. To use this option, you must have the Intel® Trace Collector installed and set up through one of its set-up scripts. This tool is a component of the Intel® Trace Analyzer and Collector.

This option provides a flexible and convenient way of instrumenting functions of a compiled application. For every function, the entry and exit points are instrumented at compile time to let the Intel® Trace Collector record functions beyond the default MPI calls. For non-MPI applications (for example, threaded or serial), you must ensure that the Intel® Trace Collector is properly initialized (VT_initialize/VT_init).

CAUTION. Be careful with full instrumentation because this feature can produce very large trace files.

For more details, see the Intel® Trace Collector User Guide.

Alternate Options

None

See Also

-
-
- tcollect-filter, Qtcollect-filter

tcollect-filter, Qtcollect-filter

Lets you enable or disable the instrumentation of specified functions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tcollect-filter` *file*

Mac OS X:

`None`

Windows:

`/Qtcollect-filter:`*file*

Arguments

file	Is a configuration file that lists filters, one per line. Each filter consists of a regular expression string and a switch. Strings with leading or trailing white spaces must be quoted. Other strings do not have to be quoted. The switch value can be ON, on, OFF, or off.

Default

OFF	Functions are not instrumented. However, if option `-tcollect` (Linux) or `/Qtcollect` (Windows) is specified, the filter setting is ".* ON" and all functions get instrumented.

Description

This option lets you enable or disable the instrumentation of specified functions.

During instrumentation, the regular expressions in the file are matched against the function names. The switch specifies whether matching functions are to be instrumented or not. Multiple filters are evaluated from top to bottom with increasing precedence.

The names of the functions to match against are formatted as follows:

- The source file name is followed by a colon-separated function name. Source file names should contain the full path, if available. For example:
  `/home/joe/src/file.f:FOO_bar`
- Classes and function names are separated by double colons. For example:
  `/home/joe/src/file.fpp:app::foo::bar`

You can use option `-opt-report` (Linux) or `/Qopt-report` (Windows) to get a full list of file and function names that the compiler recognizes from the compilation unit. This list can be used as the basis for filtering in the configuration file.

To use this option, you must have the Intel® Trace Collector installed and set up through one of its set-up scripts. This tool is a component of the Intel® Trace Analyzer and Collector.

For more details, see the Intel® Trace Collector User Guide.

Alternate Options

None

Consider the following filters in a configuration file:

```
'.*' OFF '.*vector.*' ON
```

The above will cause instrumentation of only those functions having the string 'vector' in their names. No other function will be instrumented. Note that reversing the order of the two lines will prevent instrumentation of all functions.

To get a list of the file or routine strings that can be matched by the regular expression filters, generate an optimization report with tcollect information. For example:

```
Windows OS: ifort /Qtcollect /Qopt-report /Qopt-report-phase tcollect

Linux OS: ifort -tcollect -opt-report -opt-report-phase tcollect
```

See Also

-
-
- tcollect, Qtcollect

tprofile, Qtprofile

Generates instrumentation to analyze multi-threading performance.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tprofile`

Mac OS X:

`None`

Windows:

`/Qtprofile`

Arguments

None

Default

OFF	Instrumentation is not generated by the compiler for analysis by Intel® Thread Profiler.

Description

This option generates instrumentation to analyze multi-threading performance.

To use this option, you must have Intel® Thread Profiler installed, which is one of the Intel® Threading Analysis Tools. If you do not have this tool installed, the compilation will fail. Remove the `-tprofile` (Linux) or `/Qtprofile` (Windows) option from the command line and recompile.

For more information about Intel® Thread Profiler (including an evaluation copy), open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

ftrapuv, Qtrapuv

Initializes stack local variables to an unusual value to aid error detection.

IDE Equivalent

Windows: **Data > Initialize stack variables to an unusual value**

Linux: None

Mac OS X: **Run-Time > Initialize Stack Variables to an Unusual Value**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-ftrapuv`

Windows:

`/Qtrapuv`

Arguments

None

Default

OFF	The compiler does not initialize local variables.

Description

This option initializes stack local variables to an unusual value to aid error detection. Normally, these local variables should be initialized in the application.

The option sets any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors.

This option sets option `-g` (Linux and Mac OS X) and `/Zi` or `/Z7` (Windows).

Alternate Options

None

See Also

-
-
- `g`, `Zi`, `Z7`

unroll, Qunroll

Tells the compiler the maximum number of times to unroll loops.

IDE Equivalent

Windows: **Optimization > Loop Unroll Count**

Linux: None

Mac OS X: **Optimization > Loop Unroll Count**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-unroll[=n]`

Windows:

`/Qunroll[:n]`

Arguments

n	Is the maximum number of times a loop can be unrolled. To disable loop enrolling, specify 0. On systems using IA-64 architecture, you can only specify a value of 0.

Default

`-unroll` or `/Qunroll`	The compiler uses default heuristics when unrolling loops.

Description

This option tells the compiler the maximum number of times to unroll loops.

If you do not specify *n*, the optimizer determines how many times loops can be unrolled.

Alternate Options

Linux and Mac OS X: `-funroll-loops`

Windows: `/unroll`

See Also

-
-

Optimizing Applications: Loop Unrolling

unroll-aggressive, Qunroll-aggressive

Determines whether the compiler uses more aggressive unrolling for certain loops.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-unroll-aggressive`

`-no-unroll-aggressive`

Windows:

`/Qunroll-aggressive`

`/Qunroll-aggressive-`

Arguments

None

Default

`-no-unroll-aggressive` or `/Qunroll-aggressive-`	The compiler uses default heuristics when unrolling loops.

Description

This option determines whether the compiler uses more aggressive unrolling for certain loops. The positive form of the option may improve performance.

On IA-32 architecture and Intel® 64 architecture, this option enables aggressive, complete unrolling for loops with small constant trip counts.

On IA-64 architecture, this option enables additional complete unrolling for loops that have multiple exits or outer loops that have a small constant trip count.

Alternate Options

None

uppercase, Quppercase

See names.

use-asm, Quse-asm

Tells the compiler to produce objects through the assembler.

IDE Equivalent

None

Architectures

`-use-asm`: IA-32 architecture, Intel® 64 architecture, IA-64 architecture

`/Quse-asm`: IA-64 architecture

Syntax

Linux and Mac OS X:

```
-use-asm
```

```
-no-use-asm
```

Windows:

```
/Quse-asm
```

```
/Quse-asm-
```

Arguments

None

Default

`-no-use-asm` or `/Quse-asm-`	The compiler produces objects directly.

Description

This option tells the compiler to produce objects through the assembler.

Alternate Options

None

Quse-msasm-symbols

Tells the compiler to use a dollar sign ("$") when producing symbol names.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Quse-msasm-symbols`

Arguments

None

Default

OFF	The compiler uses a period (".") when producing symbol names

Description

This option tells the compiler to use a dollar sign ("$") when producing symbol names.

Use this option if you require symbols in your .asm files to contain characters that are accepted by the MS assembler.

Alternate Options

None

Quse-vcdebug

Tells the compiler to issue debug information compatible with the Visual C++ debugger.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Quse-vcdebug`

Arguments

None

Default

OFF	Debug information is issued that is compatible with Fortran debuggers.

Description

This option tells the compiler to issue debug information compatible with the Visual C++ debugger. It prevents the compiler from issuing the extended information used by Fortran debuggers.

Alternate Options

None

Qvc

Specifies compatibility with Microsoft Visual C++ or Microsoft* Visual Studio.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Qvc7.1`

`/Qvc8`

`/Qvc9`

Arguments

None

Default

varies	When the compiler is installed, it detects which version of Visual Studio is on your system. `Qvc` defaults to the form of the option that is compatible with that version. When multiple versions of

Visual Studio are installed, the compiler installation lets you select which version you want to use. In this case, `Qvc` defaults to the version you choose.

Description

This option specifies compatibility with Visual C++ or Visual Studio.

Option	Description
`/Qvc7.1`	Specifies compatibility with Microsoft* Visual Studio .NET 2003.
`/Qvc8`	Specifies compatibility with Microsoft* Visual Studio 2005.
`/Qvc9`	Specifies compatibility with Microsoft* Visual Studio 2008.

Alternate Options

None

vec, Qvec

Enables or disables vectorization and transformations enabled for vectorization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-vec
-no-vec
```

Windows:

`/Qvec`

`/Qvec-`

Arguments

None

Default

`-vec` or `/Qvec`	Vectorization is enabled.

Description

This option enables or disables vectorization and transformations enabled for vectorization.

To disable vectorization and transformations enabled for vectorization, specify `-no-vec` (Linux and Mac OS X) or `/Qvec-` (Windows).

Alternate Options

None

See Also

-
-
- `ax, Qax`
- `x, Qx`
- `vec-report, Qvec-report`
- `vec-guard-write, Qvec-guard-write`
- `vec-threshold, Qvec-threshold`

vec-guard-write, Qvec-guard-write

Tells the compiler to perform a conditional check in a vectorized loop.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-guard-write`

`-no-vec-guard-write`

Windows:

`/Qvec-guard-write`

`/Qvec-guard-write-`

Arguments

None

Default

`-no-vec-guard-write` or `/Qvec-guard-write-`	The compiler uses default heuristics when checking vectorized loops.

Description

This option tells the compiler to perform a conditional check in a vectorized loop. This checking avoids unnecessary stores and may improve performance.

Alternate Options

None

vec-report, Qvec-report

Controls the diagnostic information reported by the vectorizer.

IDE Equivalent

Windows: **Compilation Diagnostics > Vectorizer Diagnostic Level**

Linux: None

Mac OS X: **Diagnostics > Vectorizer Diagnostic Report**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-report[n]`

Windows:

`/Qvec-report[n]`

Arguments

n	Is a value denoting which diagnostic messages to report. Possible values are:
0	Tells the vectorizer to report no diagnostic information.
1	Tells the vectorizer to report on vectorized loops.
2	Tells the vectorizer to report on vectorized and non-vectorized loops.
3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
4	Tells the vectorizer to report on non-vectorized loops.
5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.

Default

`-vec-report1` or `/Qvec-report1`	If the vectorizer has been enabled and you do not specify *n*, the compiler reports diagnostics on vectorized loops. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the diagnostic information reported by the vectorizer. The vectorizer report is sent to stdout.

If you do not specify *n*, it is the same as specifying `-vec-report1` (Linux and Mac OS X) or `/Qvec-report1` (Windows).

The vectorizer is enabled when certain compiler options are specified, such as option `-ax` or `-x` (Linux and Mac OS X), option `/Qax` or `/Qx` (Windows), option `-arch SSE` or `-arch SSE2` (Linux and Mac OS X), option `/architecture:SSE` or `/architecture:SSE2` (Windows).

If this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

vec-threshold, Qvec-threshold

Sets a threshold for the vectorization of loops.

IDE Equivalent

Windows: **Optimization > Threshold For Vectorization**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-threshold[`*n*`]`

Windows:

`/Qvec-threshold[[:]`*n*`]`

Arguments

n	Is an integer whose value is the threshold for the vectorization of loops. Possible values are 0 through 100. If *n* is 0, loops get vectorized always, regardless of computation work volume. If *n* is 100, loops get vectorized when performance gains are predicted based on the compiler analysis data. Loops get vectorized only if profitable vector-level parallel execution is almost certain. The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to vectorize only if there is a 50% probability of the code speeding up if executed in vector form.

Default

`-vec-threshold100` or `/Qvec-threshold100`	Loops get vectorized only if profitable vector-level parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the vectorization of loops based on the probability of profitable execution of the vectorized loop in parallel.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Alternate Options

None

x, Qx

Tells the compiler to generate optimized code specialized for the Intel processor that executes your program.

IDE Equivalent

Windows: **Code Generation > Intel Processor-Specific Optimization**

Optimization > Use Intel(R) Processor Extensions

Linux: None

Mac OS X: **Code Generation > Intel Processor-Specific Optimization**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

-xprocessor

Windows:

/Qxprocessor

Arguments

processor Indicates the processor for which code is generated. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`Host` Can generate instructions for the highest instruction set available on the compilation host processor.
On Intel processors, this may correspond to the most suitable `-x` (Linux* and Mac OS* X) or `/Qx` (Windows*) option. On non-Intel processors, this may correspond to the most suitable `-m` (Linux and Mac OS X) or `/arch` (Windows) option.

	The resulting executable may not run on a processor different from the host in the following cases: • If the processor does not support all of the instructions supported by the host processor. • If the host is an Intel processor and the other processor is a non-Intel processor.
`AVX`	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).
`SSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
`SSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.
`SSE3_ATOM`	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology. Can generate MOVBE instructions, depending on the setting of option `-minstruction` (Linux and Mac OS) or `/Qinstruction` (Windows).

SSSE3	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. For Mac OS* X systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
SSE3	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. For Mac OS* X systems, this value is only supported on IA-32 architecture.This replaces value P, which is deprecated.
SSE2	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. This value is not available on Mac OS* X systems. This replaces value N, which is deprecated.

Default

Windows* systems: None Linux* systems: None Mac OS* X systems using IA-32 architecture: SSE3 Mac OS* X systems using Intel® 64 architecture: SSSE3	On Windows systems, if neither /Qx nor /arch is specified, the default is /arch:SSE2. On Linux systems, if neither -x nor -m is specified, the default is -msse2.

Description

This option tells the compiler to generate optimized code specialized for the Intel processor that executes your program. It also enables optimizations in addition to Intel processor-specific optimizations. The specialized code generated by this option may run only on a subset of Intel processors.

This option can enable optimizations depending on the argument specified. For example, it may enable Intel® Streaming SIMD Extensions 4 (Intel® SSE4), Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Intel® Streaming SIMD Extensions 2 (Intel® SSE2), or Intel® Streaming SIMD Extensions (Intel® SSE) instructions.

The binaries produced by these values will run on Intel processors that support all of the features for the targeted processor. For example, binaries produced with SSE3 will run on an Intel® Core™ 2 Duo processor, because that processor completely supports all of the capabilities of the Intel® Pentium® 4 processor, which the SSE3 value targets. Specifying the SSSE3 value has the potential of using more features and optimizations available to the Intel® Core™ 2 Duo processor.

Do not use *processor* values to create binaries that will execute on a processor that is not compatible with the targeted processor. The resulting program may fail with an illegal instruction exception or display other unexpected behavior. For example, binaries produced with SSE3 may produce code that will not run on Intel® Pentium® III processors or earlier processors that do not support SSE3 instructions.

Compiling the main program with any of the *processor* values produces binaries that display a fatal run-time error if they are executed on unsupported processors. For more information, see *Optimizing Applications*.

If you specify more than one *processor* value, code is generated for only the highest-performing processor specified. The highest-performing to lowest-performing *processor* values are: SSE4.2, SSE4.1, SSSE3, SSE3, SSE2. Note that *processor* values AVX and SSE3_ATOM do not fit within this group.

Compiler options `m` and `arch` produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel processors.

Previous value O is deprecated and has been replaced by option `-msse3` (Linux and Mac OS X) and option `/arch:SSE3` (Windows).

Previous values W and K are deprecated. The details on replacements are as follows:

- Mac OS X systems: On these systems, there is no exact replacement for W or K. You can upgrade to the default option `-msse3` (IA-32 architecture) or option `-mssse3` (Intel® 64 architecture).
- Windows and Linux systems: The replacement for W is `-msse2` (Linux) or `/arch:SSE2` (Windows). There is no exact replacement for K. However, on Windows systems, `/QxK` is interpreted as `/arch:IA32`; on Linux systems, `-xK` is interpreted as `-mia32`. You can also do one of the following:

- Upgrade to option -msse2 (Linux) or option /arch:SSE2 (Windows). This will produce one code path that is specialized for Intel® SSE2. It will not run on earlier processors
- Specify the two option combination -mia32 -axSSE2 (Linux) or /arch:IA32 /QaxSSE2 (Windows). This combination will produce an executable that runs on any processor with IA-32 architecture but with an additional specialized Intel® SSE2 code path.

The -x and /Qx options enable additional optimizations not enabled with option -m or option /arch.

On Windows* systems, options /Qx and /arch are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning. Similarly, on Linux* and Mac OS* X systems, options -x and -m are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Alternate Options

None

See Also

-
-
- ax, Qax
- m
- arch
- minstruction, Qinstruction

zero, Qzero

Initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.

IDE Equivalent

Windows: **Data > Initialize Local Saved Scalars to Zero**

Linux: None

Mac OS X: **Data > Initialize Local Saved Scalars to Zero**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-zero`

`-nozero`

Windows:

`/Qzero`

`/Qzero-`

Arguments

None

Default

`-nozero` or `/Qzero-` Local scalar variables are not initialized to zero.

Description

This option initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.

Use `-save` (Linux and Mac OS X) or `/Qsave` (Windows) on the command line to make all local variables specifically marked as SAVE.

Alternate Options

None

See Also

-
-
- `save`

r8, r16

See real-size.

rcd, Qrcd

Enables fast float-to-integer conversions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-rcd`

Windows:

`/Qrcd`

Arguments

None

Default

OFF	Floating-point values are truncated when a conversion to an integer is involved. On Windows, this is the same as specifying `/QIfist-`.

Description

This option enables fast float-to-integer conversions. It can improve the performance of code that requires floating-point-to-integer conversions.

The system default floating-point rounding mode is round-to-nearest. However, the Fortran language requires floating-point values to be truncated when a conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating-point-to-integer conversion and change it back afterwards.

This option disables the change to truncation of the rounding mode for all floating-point calculations, including floating point-to-integer conversions. This option can improve performance, but floating-point conversions to integer will not conform to Fortran semantics.

Alternate Options

Linux and Mac OS X: None

Windows: `/QIfist`

rct, Qrct

Sets the internal FPU rounding control to Truncate.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-rct`

Windows:

`/Qrct`

Arguments

None

Default

OFF	The compiler uses the default setting for the FPU rounding control.

Description

This option sets the internal FPU rounding control to Truncate.

Alternate Options

Linux and Mac OS X: None

Windows: `/rounding-mode:chopped`

real-size

Specifies the default KIND for real and complex variables.

IDE Equivalent

Windows: **Data > Default Real KIND**

Linux: None

Mac OS X: **Data > Default Real KIND**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-real-size` *`size`*

Windows:

`/real-size:`*`size`*

Arguments

`size`	Is the size for real and complex variables. Possible values are: 32, 64, or 128.

Default

`real-size 32`	Default real and complex variables are 4 bytes long (REAL(KIND=4) and COMPLEX(KIND=4)).

Description

This option specifies the default size (in bits) for real and complex variables.

Option	Description
`real-size 32`	Makes default real and complex variables 4 bytes long. REAL declarations are treated as single precision REAL (REAL(KIND=4)) and COMPLEX declarations are treated as COMPLEX (COMPLEX(KIND=4)).
`real-size 64`	Makes default real and complex variables 8 bytes long. REAL declarations are treated as DOUBLE PRECISION (REAL(KIND=8)) and COMPLEX declarations are treated as DOUBLE COMPLEX (COMPLEX(KIND=8)).
`real-size 128`	Makes default real and complex variables 16 bytes long. REAL declarations are treated as extended precision REAL (REAL(KIND=16)); COMPLEX and DOUBLE COMPLEX declarations are treated as extended precision COMPLEX (COMPLEX(KIND=16)).

These compiler options can affect the result type of intrinsic procedures, such as CMPLX, FLOAT, REAL, SNGL, and AIMAG, which normally produce single-precision REAL or COMPLEX results. To prevent this effect, you must explicitly declare the kind type for arguments of such intrinsic procedures.

For example, if `real-size 64` is specified, the CMPLX intrinsic will produce a result of type DOUBLE COMPLEX (COMPLEX(KIND=8)). To prevent this, you must explicitly declare any real argument to be REAL(KIND=4), and any complex argument to be COMPLEX(KIND=4).

Alternate Options

`real-size 64`	Linux and Mac OS X: `-r8`, `-autodouble` Windows: `/4R8`, `/Qautodouble`
`real-size 128`	Linux and Mac OS X: `-r16` Windows: `/4R16`

recursive

Tells the compiler that all routines should be compiled for possible recursive execution.

IDE Equivalent

Windows: **Code Generation > Enable Recursive Routines**

Linux: None

Mac OS X: **Code Generation > Enable Recursive Routines**

Architectures

IA-32, Intel® 64, IA-64 architectures

Systems: Windows, Linux

Syntax

Linux and Mac OS X:

`-recursive`

`-norecursive`

Windows:

`/recursive`

`/norecursive`

Arguments

None

Default

`norecursive`	Routines are not compiled for possible recursive execution.

Description

This option tells the compiler that all routines should be compiled for possible recursive execution. It sets the `automatic` option.

Alternate Options

None

See Also

-
- `automatic`

reentrancy

Tells the compiler to generate reentrant code to support a multithreaded application.

IDE Equivalent

Windows: **Code Generation > Generate Reentrant Code**

Linux: None

Mac OS X: **Code Generation > Generate Reentrant Code**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-reentrancy` *`keyword`*

`-noreentrancy`

Windows:

`/reentrancy:`*`keyword`*

`/noreentrancy`

Arguments

`keyword`	Specifies details about the program. Possible values are:	
	`none`	Tells the run-time library (RTL) that the program does not rely on threaded or asynchronous reentrancy. The RTL will not guard against such interrupts inside its own critical regions. This is the same as specifying noreentrancy.
	`async`	Tells the run-time library (RTL) that the program may contain asynchronous (AST) handlers that could call the RTL. This causes the RTL to guard against AST interrupts inside its own critical regions.

`threaded`	Tells the run-time library (RTL) that the program is multithreaded, such as programs using the POSIX threads library. This causes the RTL to use thread locking to guard its own critical regions.

Default

`noreentrancy`	The compiler does not generate reentrant code for applications.

Description

This option tells the compiler to generate reentrant code to support a multithreaded application.

If you do not specify a keyword for reentrancy, it is the same as specifying `reentrancy threaded`.

Note that if option threads is specified, it sets option `reentrancy threaded`, since multithreaded code must be reentrant.

Alternate Options

None

See Also

-
- `threads`

RTCu

See check.

S

Causes the compiler to compile to an assembly file only and not link.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-S`

Windows:

`/S`

Arguments

None

Default

OFF	Normal compilation and linking occur.

Description

This option causes the compiler to compile to an assembly file only and not link.

On Linux and Mac OS X systems, the assembly file name has a .s suffix. On Windows systems, the assembly file name has an .asm suffix.

Alternate Options

Linux and Mac OS X: None

Windows: `/Fa`, `/asmfile`

See Also

-
- Fa

safe-cray-ptr, Qsafe-cray-ptr

Tells the compiler that Cray pointers do not alias other variables.*

IDE Equivalent

Windows: **Data > Assume Cray Pointers Do Not Share Memory Locations**

Linux: None

Mac OS X: **Data > Assume Cray Pointers Do Not Share Memory Locations**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-safe-cray-ptr
```

Windows:

```
/Qsafe-cray-ptr
```

Arguments

None

Default

OFF	The compiler assumes that Cray pointers alias other variables.

Description

This option tells the compiler that Cray pointers do not alias (that is, do not specify sharing memory with) other variables.

Alternate Options

None

Example

Consider the following:

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
b(i) = a(i) + 1
enddo
```

By default, the compiler assumes that b and a are aliased. To prevent such an assumption, specify the `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) option, and the compiler will treat b(i) and a(i) as independent of each other.

However, if the variables are intended to be aliased with Cray pointers, using the option produces incorrect results. In the following example, you should not use the option:

```
pointer (pb, b)
pb = loc(a(2))
do i=1, n
b(i) = a(i) +1
enddo
```

save, Qsave

Causes variables to be placed in static memory.

IDE Equivalent

Windows: **Data > Local Variable Storage**

Linux: None

Mac OS X: **Data > Local Variable Storage**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-save`

Windows:

`/Qsave`

Arguments

None

Default

`-auto-scalar` or `/Qauto-scalar`	Scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL are allocated to the run-time stack. Note that if option `recursive`, `-openmp` (Linux and Mac OS X), or `/Qopenmp` (Windows) is specified, the default is `-automatic` (Linux) or `/Qauto` (Windows).

Description

This option saves all variables in static allocation except local variables within a recursive routine and variables declared as AUTOMATIC.

If you want all local, non-SAVEd variables to be allocated to the run-time stack, specify option `automatic`.

Alternate Options

Linux and Mac OS X: `-noautomatic`, `-noauto`

Windows: `/noautomatic`, `/noauto`, `/4Na`

See Also

-
-
- `automatic`
- `auto_scalar`

save-temps, Qsave-temps

Tells the compiler to save intermediate files created during compilation.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-save-temps`

`-no-save-temps`

Windows:

`/Qsave-temps`

`/Qsave-temps-`

Arguments

None

Default

Linux and Mac OS X: `-no-save-temps` Windows: .obj files are saved	On Linux and Mac OS X systems, the compiler deletes intermediate files after compilation is completed. On Windows systems, the compiler saves only intermediate object files after compilation is completed.

Description

This option tells the compiler to save intermediate files created during compilation. The names of the files saved are based on the name of the source file; the files are saved in the current working directory.

If `-save-temps` or `/Qsave-temps` is specified, the following occurs:

- The object .o file (Linux and Mac OS X) or .obj file (Windows) is saved.
- The assembler .s file (Linux and Mac OS X) or .asm file (Windows) is saved if you specified -use-asm (Linux or Mac OS X) or /Quse-asm (Windows).
- The .i or .i90 file is saved if the fpp preprocessor is invoked.

If `-no-save-temps` is specified on Linux or Mac OS X systems, the following occurs:

- The .o file is put into `/tmp` and deleted after calling `ld`.
- The preprocessed file is not saved after it has been used by the compiler.

If `/Qsave-temps-` is specified on Windows systems, the following occurs:

- The .obj file is not saved after the linker step.
- The preprocessed file is not saved after it has been used by the compiler.

NOTE. This option only saves intermediate files that are normally created during compilation.

Alternate Options

None

Example

If you compile program `my_foo.F` on a Linux or Mac OS X system and you specify option `-save-temps` and option `-use-asm`, the compilation will produce `files my_foo.o`, `my_foo.s`, and `my_foo.i`.

If you compile program `my_foo.fpp` on a Windows system and you specif option `/Qsave-temps` and option `/Quse-asm`, the compilation will produce `files my_foo.obj`, `my_foo.asm`, and `my_foo.i`.

scalar-rep, Qscalar-rep

Enables scalar replacement performed during loop transformation.

IDE Equivalent

None

Architectures

IA-32 architecture

Syntax

Linux and Mac OS X:

`-scalar-rep`

`-no-scalar-rep`

Windows:

`/Qscalar-rep`

`/Qscalar-rep-`

Arguments

None

Default

`-no-scalar-rep` or `/Qscalar-rep-`	Scalar replacement is not performed during loop transformation.

Description

This option enables scalar replacement performed during loop transformation. To use this option, you must also specify `O3`.

Alternate Options

None

See Also

-
-
- O

shared

Tells the compiler to produce a dynamic shared object instead of an executable.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-shared`

Mac OS X:

None

Windows:

None

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option tells the compiler to produce a dynamic shared object (DSO) instead of an executable. This includes linking in all libraries dynamically and passing `-shared` to the linker.

On systems using IA-32 architecture and Intel® 64 architecture, you must specify option `fpic` for the compilation of each object file you want to include in the shared library.

Alternate Options

None

See Also

-
- `dynamiclib`
- `fpic`
- `Xlinker`

shared-intel

Causes Intel-provided libraries to be linked in dynamically.

IDE Equivalent

Windows: None

Linux: None

Mac OS X: **Run-Time > Intel Runtime Libraries**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-shared-intel`

Windows:

`None`

Arguments

None

Default

OFF	Intel libraries are linked in statically, with the exception of libguide on Linux* and Mac OS* X systems, where it is linked in dynamically.

Description

This option causes Intel-provided libraries to be linked in dynamically. It is the opposite of `-static-intel`.

NOTE. On Mac OS X systems, when you set "Intel Runtime Libraries" to "Dynamic", you must also set the DYLD_LIBRARY_PATH environment variable within Xcode or an error will be displayed.

Alternate Options

Linux and Mac OS X: `-i-dynamic` (this is a deprecated option)

Windows: None

See Also

-
- `static-intel`

shared-libgcc

Links the GNU `libgcc` *library dynamically.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-shared-libgcc`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

`-shared-libgcc`	The compiler links the `libgcc` library dynamically.

Description

This option links the GNU `libgcc` library dynamically. It is the opposite of option `static-libgcc`.

This option is useful when you want to override the default behavior of the `static` option, which causes all libraries to be linked statically.

Alternate Options

None

See Also

-
- static-libgcc

source

Tells the compiler to compile the file as a Fortran source file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/source:`*`file`*

Arguments

`file`	Is the name of the file.

Default

OFF	Files that do not end in standard Fortran file extensions are not compiled as Fortran files.

Description

This option tells the compiler to compile the file as a Fortran source file.

This option is useful when you have a Fortran file with a nonstandard file extension (that is, not one of .F, .FOR, or .F90).

This option assumes the file specified uses fixed source form. If the file uses free source form, you must also specify option `free`.

Alternate Options

Linux and Mac OS X: `-Tf file`

Windows: `/Tf file`

See Also

-
- `extfor`
- `free`

sox, Qsox

Tells the compiler to save the compilation options and version number in the Linux OS executable or the Windows* OS object file.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

```
-sox
-no-sox
```

Mac OS X:

```
None
```

Windows:

```
/Qsox
/Qsox-
```

Arguments

None

Default

`-no-sox` or `/Qsox-`	The compiler does not save the compiler options and version number in the executable.

Description

Tells the compiler to save the compilation options and version number in the Linux* OS executable or the Windows* OS object file.

On Linux systems, the size of the executable on disk is increased slightly by the inclusion of these infotmation strings.

This option forces the compiler to embed in each object file or assembly output a string that contains information about the compiler version and compilation options for each source file that has been compiled.

On Windows systems, the information stays in the object file. On Linux systems, when you link the object files into an executable file, the linker places each of the information strings into the header of the executable. It is then possible to use a tool, such as a strings utility, to determine what options were used to build the executable file.

If `-no-sox` or `/Qsox-` is specified, this extra information is not put into the object or assembly output generated by the compiler.

Alternate Options

None

stand

Tells the compiler to issue compile-time messages for nonstandard language elements.

IDE Equivalent

Windows: **Compilation Diagnostics > Warn For Nonstandard Fortran**

Linux: None

Mac OS X: **Compiler Diagnostics > Warn For Nonstandard Fortran**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-stand [keyword]`

`-nostand`

Windows:

`/stand[:keyword]`

`/nostand`

Arguments

keyword	Specifies the language to use as the standard. Possible values are:
`none`	Issue no messages for nonstandard language elements.
`f90`	Issue messages for language elements that are not standard in Fortran 90.
`f95`	Issue messages for language elements that are not standard in Fortran 95.
`f03`	Issue messages for language elements that are not standard in Fortran 2003.

Default

`nostand`	The compiler issues no messages for nonstandard language elements.

Description

This option tells the compiler to issue compile-time messages for nonstandard language elements.

If you do not specify a keyword for `stand`, it is the same as specifying `stand f95`.

Option	Description
`stand none`	Tells the compiler to issue no messages for nonstandard language elements. This is the same as specifying `nostand`.

Option	Description
`stand f90`	Tells the compiler to issue messages for language elements that are not standard in Fortran 90.
`stand f95`	Tells the compiler to issue messages for language elements that are not standard in Fortran 95.
`stand f03`	Tells the compiler to issue messages for language elements that are not standard in Fortran 2003. This option is set if you specify `warn stderrors`.

Alternate Options

`stand none`	Linux and Mac OS X: `-nostand` Windows: `/nostand`, `/4Ns`
`stand f90`	Linux and Mac OS X: `-std90` Windows: `/4Ys`
`stand f95`	Linux and Mac OS X: `-std95` Windows: None
`stand f03`	Linux and Mac OS X: `-std03`, `-stand`, `-std` Windows: `/stand`

See Also

-
- `warn stderrors`

static

Prevents linking with shared libraries.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-static`

Mac OS X:

`None`

Windows:

`/static`

Arguments

None

Default

`static`	The compiler does not link with shared libraries.

Description

This option prevents linking with shared libraries. It causes the executable to link all libraries statically.

Alternate Options

None

staticlib

Invokes the `libtool` *command to generate static libraries.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux:

`None`

Mac OS X:

`-staticlib`

Windows:

`None`

Arguments

None

Default

OFF	The compiler produces an executable.

Description

This option invokes the `libtool` command to generate static libraries.

When passed this option, the compiler uses the `libtool` command to produce a static library instead of an executable when linking.

To build dynamic libraries, you should specify option `-dynamiclib` or `libtool -dynamic <objects>`.

Alternate Options

None

See Also

-
- dynamiclib

static-intel

Causes Intel-provided libraries to be linked in statically.

IDE Equivalent

Windows: None

Linux: None

Mac OS X: **Run-Time > Intel Runtime Libraries**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-static-intel`

Windows:

`None`

Arguments

None

Default

OFF	Intel libraries are linked in statically, with the exception of libguide, which is linked in dynamically.

Description

This option causes Intel-provided libraries to be linked in statically. It is the opposite of `-shared-intel`.

Alternate Options

Linux and Mac OS X: `i-static` (this is a deprecated option)

Windows: None

See Also

-
- `shared-intel`

static-libgcc

Links the GNU `libgcc` *library statically.*

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-static-libgcc`

Mac OS X:

`None`

Windows:

`None`

Arguments

None

Default

OFF	DEFAULT_DESC

Description

This option links the GNU `libgcc` library statically. It is the opposite of option `libgcc`.

This option is useful when you want to override the default behavior of the `libgcc` option, which causes all libraries to be linked statically.

Alternate Options

None

See Also

-
- shared-libgcc

std, std90, std95, std03

See stand.

std, std90, std95, std03

See stand.

std, std90, std95, std03

See stand.

std, std90, std95, std03

See stand.

syntax-only

Tells the compiler to check only for correct syntax.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-syntax-only
```

Windows:

```
/syntax-only
```

Arguments

None

Default

OFF	Normal compilation is performed.

Description

This option tells the compiler to check only for correct syntax. It lets you do a quick syntax check of your source file.

Compilation stops after the source file has been parsed. No code is generated, no object file is produced, and some error checking done by the optimizer is bypassed.

Warnings and messages appear on `stderr`.

Alternate Options

Linux: `-y, -fsyntax-only, -syntax` (this is a deprecated option)

Mac OS X: `-y, -fsyntax-only`

Windows: `/Zs`

T

Tells the linker to read link commands from a file.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-T`*`file`*

Mac OS X:

None

Windows:

None

Arguments

file	Is the name of the file.

Default

OFF	The linker does not read link commands from a file.

Description

This option tells the linker to read link commands from a file.

Alternate Options

None

tcheck, Qtcheck

Enables analysis of threaded applications.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

-tcheck

Mac OS X:

None

Windows:

`/Qtcheck`

Arguments

None

Default

OFF	Threaded applications are not instrumented by the compiler for analysis by Intel® Thread Checker.

Description

This option enables analysis of threaded applications.

To use this option, you must have Intel® Thread Checker installed, which is one of the Intel® Threading Analysis Tools. If you do not have this tool installed, the compilation will fail. Remove the `-tcheck` (Linux) or `/Qtcheck` (Windows) option from the command line and recompile.

For more information about Intel® Thread Checker (including an evaluation copy), open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

tcollect, Qtcollect

Inserts instrumentation probes calling the Intel® Trace Collector API.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tcollect[`*`lib`*`]`

Mac OS X:

None

Windows:

/Qtcollect[:*lib*]

Arguments

lib	Is one of the Intel® Trace Collector libraries; for example, VT, VTcs, VTmc, or VTfs. If you do not specify *lib*, the default library is VT.

Default

OFF	Instrumentation probes are not inserted into compiled applications.

Description

This option inserts instrumentation probes calling the Intel® Trace Collector API. To use this option, you must have the Intel® Trace Collector installed and set up through one of its set-up scripts. This tool is a component of the Intel® Trace Analyzer and Collector.

This option provides a flexible and convenient way of instrumenting functions of a compiled application. For every function, the entry and exit points are instrumented at compile time to let the Intel® Trace Collector record functions beyond the default MPI calls. For non-MPI applications (for example, threaded or serial), you must ensure that the Intel® Trace Collector is properly initialized (VT_initialize/VT_init).

CAUTION. Be careful with full instrumentation because this feature can produce very large trace files.

For more details, see the Intel® Trace Collector User Guide.

Alternate Options

None

See Also

-
-

- tcollect-filter, Qtcollect-filter

tcollect-filter, Qtcollect-filter

Lets you enable or disable the instrumentation of specified functions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tcollect-filter` *file*

Mac OS X:

`None`

Windows:

`/Qtcollect-filter:`*file*

Arguments

file	Is a configuration file that lists filters, one per line. Each filter consists of a regular expression string and a switch. Strings with leading or trailing white spaces must be quoted. Other strings do not have to be quoted. The switch value can be ON, on, OFF, or off.

Default

OFF	Functions are not instrumented. However, if option `-tcollect` (Linux) or `/Qtcollect` (Windows) is specified, the filter setting is ".* ON" and all functions get instrumented.

Description

This option lets you enable or disable the instrumentation of specified functions.

During instrumentation, the regular expressions in the file are matched against the function names. The switch specifies whether matching functions are to be instrumented or not. Multiple filters are evaluated from top to bottom with increasing precedence.

The names of the functions to match against are formatted as follows:

- The source file name is followed by a colon-separated function name. Source file names should contain the full path, if available. For example:
  `/home/joe/src/file.f:FOO_bar`
- Classes and function names are separated by double colons. For example:
  `/home/joe/src/file.fpp:app::foo::bar`

You can use option `-opt-report` (Linux) or `/Qopt-report` (Windows) to get a full list of file and function names that the compiler recognizes from the compilation unit. This list can be used as the basis for filtering in the configuration file.

To use this option, you must have the Intel® Trace Collector installed and set up through one of its set-up scripts. This tool is a component of the Intel® Trace Analyzer and Collector.

For more details, see the Intel® Trace Collector User Guide.

Alternate Options

None

Consider the following filters in a configuration file:

```
'.*' OFF '.*vector.*' ON
```

The above will cause instrumentation of only those functions having the string 'vector' in their names. No other function will be instrumented. Note that reversing the order of the two lines will prevent instrumentation of all functions.

To get a list of the file or routine strings that can be matched by the regular expression filters, generate an optimization report with tcollect information. For example:

```
Windows OS: ifort /Qtcollect /Qopt-report /Qopt-report-phase tcollect

Linux OS: ifort -tcollect -opt-report -opt-report-phase tcollect
```

See Also

-
-
- tcollect, Qtcollect

Tf

See source.

threads

Tells the linker to search for unresolved references in a multithreaded run-time library.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-threads`

`-nothreads`

Windows:

`/threads`

`/nothreads`

Arguments

None

Default

Systems using Intel® 64 architecture: `threads` Systems using IA-32 architecture and IA-64 architecture: `nothreads`	On systems using IA-32 architecture and IA-64 architecture, the linker does not search for unresolved references in a mutithreaded run-time library. On systems using Intel® 64 architectures, it does.

Description

This option tells the linker to search for unresolved references in a multithreaded run-time library.

This option sets option `reentrancy threaded`.

Windows systems: The following table shows which options to specify for a multithreaded run-time library.

Type of Library	Options Required	Alternate Option
Multithreaded	`/libs:static` `/threads`	`/MT`

Debug multithreaded	`/libs:static`
	`/threads`
	`/dbglibs`
		`/MTd`

Multithreaded DLLs	`/libs:dll` `/threads`	`/MD`

Multithreaded debug DLLs	`/libs:dll`
	`/threads`
	`/dbglibs`
		`/MDd`

Alternate Options

None

See Also

- Building Applications: Specifying Consistent Library Types; Programming with Mixed Languages Overview

tprofile, Qtprofile

Generates instrumentation to analyze multi-threading performance.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux:

`-tprofile`

Mac OS X:

`None`

Windows:

`/Qtprofile`

Arguments

None

Default

OFF	Instrumentation is not generated by the compiler for analysis by Intel® Thread Profiler.

Description

This option generates instrumentation to analyze multi-threading performance.

To use this option, you must have Intel® Thread Profiler installed, which is one of the Intel® Threading Analysis Tools. If you do not have this tool installed, the compilation will fail. Remove the `-tprofile` (Linux) or `/Qtprofile` (Windows) option from the command line and recompile.

For more information about Intel® Thread Profiler (including an evaluation copy), open the page associated with threading tools at Intel® Software Development Products.

Alternate Options

None

traceback

Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

IDE Equivalent

Windows: **Run-time > Generate Traceback Information**

Linux: None

Mac OS X: **Run-time > Generate Traceback Information**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-traceback`

`-notraceback`

Windows:

`/traceback`

`/notraceback`

Arguments

None

Default

`notraceback`	No extra information is generated in the object file to produce traceback information.

Description

This option tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

When the severe error occurs, source file, routine name, and line number correlation information is displayed along with call stack hexadecimal addresses (program counter trace).

Note that when a severe error occurs, advanced users can also locate the cause of the error using a map file and the hexadecimal addresses of the stack displayed when the error occurs.

This option increases the size of the executable program, but has no impact on run-time execution speeds.

It functions independently of the debug option.

On Windows systems, `traceback` sets the `/Oy-` option, which forces the compiler to use EBP as the stack frame pointer.

On Windows systems, the linker places the traceback information in the executable image, in a section named ".trace". To see which sections are in an image, use the command:

```
link -dump -summary your_app_name.exe
```

To see more detailed information, use the command:

```
link -dump -headers your_app_name.exe
```

On Windows systems, when requesting traceback, you must set Enable Incremental Linking in the VS .NET* IDE Linker Options to No. On systems using IA-32 architecture and Intel® 64 architecture, you must also set Omit Frame Pointers (the `/Oy` option) in the Optimization Options to "No."

On Linux systems, to display the section headers in the image (including the header for the .trace section, if any), use the command:

```
objdump -h your_app_name.exe
```

On Mac OS X systems, to display the section headers in the image, use the command:

```
otool -l your_app_name.exe
```

Alternate Options

None

tune

Determines the version of the architecture for which the compiler generates instructions.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-tune keyword`

Windows:

`/tune:keyword`

Arguments

keyword	Specifies the processor type. Possible values are:	
	`pn1`	Optimizes for the Intel® Pentium® processor.
	`pn2`	Optimizes for the Intel® Pentium® Pro, Intel® Pentium® II, and Intel® Pentium® III processors.
	`pn3`	Optimizes for the Intel® Pentium® Pro, Intel® Pentium® II, and Intel® Pentium® III processors. This is the same as specifying `pn2`.
	`pn4`	Optimizes for the Intel® Pentium® 4 processor.

Default

`pn4`	The compiler optimizes for the Intel® Pentium® 4 processor.

Description

This option determines the version of the architecture for which the compiler generates instructions.

On systems using Intel® 64 architecture, only *keyword* `pn4` is valid.

Alternate Options

None

u (Linux* and Mac OS* X)

See warn.

u (Windows*)

Undefines all previously defined preprocessor values.

IDE Equivalent

Windows: **Preprocessor > Undefine All Preprocessor Definitions**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
None
```

Windows:

```
/u
```

Arguments

None

Default

OFF	Defined preprocessor values are in effect until they are undefined.

Description

This option undefines all previously defined preprocessor values.

To undefine specific preprocessor values, use the `/U` option.

Alternate Options

None

See Also

-
- U

U

Undefines any definition currently in effect for the specified symbol.

IDE Equivalent

Windows: **Preprocessor > Undefine Preprocessor Definitions**

Linux: None

Mac OS X: **Preprocessor > Undefine Preprocessor Definitions**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

-Uname

Windows:

/Uname

Arguments

name	Is the name of the symbol to be undefined.

Default

OFF	Symbol definitions are in effect until they are undefined.

Description

This option undefines any definition currently in effect for the specified symbol.

On Windows systems, use the `/u` option to undefine all previously defined preprocessor values.

Alternate Options

Linux and Mac OS X: None

Windows: `/undefine:`*`name`*

See Also

-
- `u` (Windows)

undefine

See U.

unroll, Qunroll

Tells the compiler the maximum number of times to unroll loops.

IDE Equivalent

Windows: **Optimization > Loop Unroll Count**

Linux: None

Mac OS X: **Optimization > Loop Unroll Count**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-unroll[=`*`n`*`]`

Windows:

`/Qunroll[:`*`n`*`]`

Arguments

n	Is the maximum number of times a loop can be unrolled. To disable loop enrolling, specify 0. On systems using IA-64 architecture, you can only specify a value of 0.

Default

`-unroll` or `/Qunroll`	The compiler uses default heuristics when unrolling loops.

Description

This option tells the compiler the maximum number of times to unroll loops.

If you do not specify *n*, the optimizer determines how many times loops can be unrolled.

Alternate Options

Linux and Mac OS X: `-funroll-loops`

Windows: `/unroll`

See Also

-
-

Optimizing Applications: Loop Unrolling

unroll-aggressive, Qunroll-aggressive

Determines whether the compiler uses more aggressive unrolling for certain loops.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-unroll-aggressive`

`-no-unroll-aggressive`

Windows:

`/Qunroll-aggressive`

`/Qunroll-aggressive-`

Arguments

None

Default

`-no-unroll-aggressive` or `/Qunroll-aggressive-`	The compiler uses default heuristics when unrolling loops.

Description

This option determines whether the compiler uses more aggressive unrolling for certain loops. The positive form of the option may improve performance.

On IA-32 architecture and Intel® 64 architecture, this option enables aggressive, complete unrolling for loops with small constant trip counts.

On IA-64 architecture, this option enables additional complete unrolling for loops that have multiple exits or outer loops that have a small constant trip count.

Alternate Options

None

uppercase, Quppercase

See names.

us

See assume.

use-asm, Quse-asm

Tells the compiler to produce objects through the assembler.

IDE Equivalent

None

Architectures

`-use-asm`: IA-32 architecture, Intel® 64 architecture, IA-64 architecture

`/Quse-asm`: IA-64 architecture

Syntax

Linux and Mac OS X:

`-use-asm`

`-no-use-asm`

Windows:

`/Quse-asm`

`/Quse-asm-`

Arguments

None

Default

`-no-use-asm` or `/Quse-asm-`	The compiler produces objects directly.

Description

This option tells the compiler to produce objects through the assembler.

Alternate Options

None

v

Specifies that driver tool commands should be displayed and executed.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-v[file]`

Windows:

`None`

Arguments

file	Is the name of a file.

Default

OFF	No tool commands are shown.

Description

This option specifies that driver tool commands should be displayed and executed.

If you use this option without specifying a file name, the compiler displays only the version of the compiler.

If you want to display processing information (pass information and source file names), specify option `watch:all`.

Alternate Options

Linux and Mac OS X: `-watch cmd`

Windows: `/watch:cmd`

See Also

-
- `dryrun`
- `watch`

V (Linux* and Mac OS* X)

See logo

V (Windows*)

See bintext.

vec, Qvec

Enables or disables vectorization and transformations enabled for vectorization.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

```
-vec
-no-vec
```

Windows:

```
/Qvec
```

/Qvec-

Arguments

None

Default

-vec or/Qvec	Vectorization is enabled.

Description

This option enables or disables vectorization and transformations enabled for vectorization.

To disable vectorization and transformations enabled for vectorization, specify -no-vec (Linux and Mac OS X) or /Qvec- (Windows).

Alternate Options

None

See Also

-
-
- ax, Qax
- x, Qx
- vec-report, Qvec-report
- vec-guard-write, Qvec-guard-write
- vec-threshold, Qvec-threshold

vec-guard-write, Qvec-guard-write

Tells the compiler to perform a conditional check in a vectorized loop.

IDE Equivalent

None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-guard-write`

`-no-vec-guard-write`

Windows:

`/Qvec-guard-write`

`/Qvec-guard-write-`

Arguments

None

Default

`-no-vec-guard-write` or `/Qvec-guard-write-`	The compiler uses default heuristics when checking vectorized loops.

Description

This option tells the compiler to perform a conditional check in a vectorized loop. This checking avoids unnecessary stores and may improve performance.

Alternate Options

None

vec-report, Qvec-report

Controls the diagnostic information reported by the vectorizer.

IDE Equivalent

Windows: **Compilation Diagnostics > Vectorizer Diagnostic Level**

Linux: None

Mac OS X: **Diagnostics > Vectorizer Diagnostic Report**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-report[n]`

Windows:

`/Qvec-report[n]`

Arguments

n	Is a value denoting which diagnostic messages to report. Possible values are:
0	Tells the vectorizer to report no diagnostic information.
1	Tells the vectorizer to report on vectorized loops.
2	Tells the vectorizer to report on vectorized and non-vectorized loops.
3	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
4	Tells the vectorizer to report on non-vectorized loops.
5	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.

Default

`-vec-report1` or `/Qvec-report1`	If the vectorizer has been enabled and you do not specify *n*, the compiler reports diagnostics on vectorized loops. If you do not specify the option on the command line, the default is to display no messages.

Description

This option controls the diagnostic information reported by the vectorizer. The vectorizer report is sent to stdout.

If you do not specify *n*, it is the same as specifying `-vec-report1` (Linux and Mac OS X) or `/Qvec-report1` (Windows).

The vectorizer is enabled when certain compiler options are specified, such as option `-ax` or `-x` (Linux and Mac OS X), option `/Qax` or `/Qx` (Windows), option `-arch SSE` or `-arch SSE2` (Linux and Mac OS X), option `/architecture:SSE` or `/architecture:SSE2` (Windows).

If this option is specified from within the IDE, the report is included in the build log if the Generate Build Logs option is selected.

Alternate Options

None

vec-threshold, Qvec-threshold

Sets a threshold for the vectorization of loops.

IDE Equivalent

Windows: **Optimization > Threshold For Vectorization**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

`-vec-threshold[`*n*`]`

Windows:

`/Qvec-threshold[[:]`*n*`]`

Arguments

n	Is an integer whose value is the threshold for the vectorization of loops. Possible values are 0 through 100. If *n* is 0, loops get vectorized always, regardless of computation work volume.

If *n* is 100, loops get vectorized when performance gains are predicted based on the compiler analysis data. Loops get vectorized only if profitable vector-level parallel execution is almost certain.
The intermediate 1 to 99 values represent the percentage probability for profitable speed-up. For example, *n*=50 directs the compiler to vectorize only if there is a 50% probability of the code speeding up if executed in vector form.

Default

`-vec-threshold100` or `/Qvec-threshold100`	Loops get vectorized only if profitable vector-level parallel execution is almost certain. This is also the default if you do not specify *n*.

Description

This option sets a threshold for the vectorization of loops based on the probability of profitable execution of the vectorized loop in parallel.

This option is useful for loops whose computation work volume cannot be determined at compile-time. The threshold is usually relevant when the loop trip count is unknown at compile-time.

The compiler applies a heuristic that tries to balance the overhead of creating multiple threads versus the amount of work available to be shared amongst the threads.

Alternate Options

None

vms

Causes the run-time system to behave like HP Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).*

IDE Equivalent

Windows: **Compatibility > Enable VMS Compatibility**

Linux: None

Mac OS X: **Compatibility > Enable VMS Compatibility**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-vms`

`-novms`

Windows:

`/vms`

`/novms`

Arguments

None

Default

`novms` The run-time system follows default Intel® Fortran behavior.

Description

This option causes the run-time system to behave like HP* Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).

It affects the following language features:

- Certain defaults

  In the absence of other options, `vms` sets the defaults as `check format` and `check output_conversion`.

- Alignment

  Option `vms` does not affect the alignment of fields in records or items in common blocks. For compatibility with HP Fortran on OpenVMS systems, use `align norecords` to pack fields of records on the next byte boundary.

- Carriage control default

  If option `vms` and option `ccdefault default` are specified, carriage control defaults to FORTRAN if the file is formatted and the unit is connected to a terminal.

- INCLUDE qualifiers

  /LIST and /NOLIST are recognized at the end of the file name in an INCLUDE statement at compile time. If the file name in the INCLUDE statement does not specify the complete path, the path used is the current directory. Note that if vms is not specified, the path used is the directory where the file that contains the INCLUDE statement resides.

- Quotation mark character

  A quotation mark (") character is recognized as starting an octal constant ("0..7) instead of a character literal ("...").

- Deleted records in relative files

  When a record in a relative file is deleted, the first byte of that record is set to a known character (currently '@'). Attempts to read that record later result in ATTACCNON errors. The rest of the record (the whole record, if vms is not specified) is set to nulls for unformatted files and spaces for formatted files.

- ENDFILE records

  When an ENDFILE is performed on a sequential unit, an actual 1-byte record containing a Ctrl/Z is written to the file. If vms is not specified, an internal ENDFILE flag is set and the file is truncated. The vms option does not affect ENDFILE on relative files: these files are truncated.

- Implied logical unit numbers

  The vms option enables Intel Fortran to recognize certain environment variables at run time for ACCEPT, PRINT, and TYPE statements and for READ and WRITE statements that do not specify a unit number (such as READ (*,1000)).

- Treatment of blanks in input

  The vms option causes the defaults for the keyword BLANK in OPEN statements to become 'NULL' for an explicit OPEN and 'ZERO' for an implicit OPEN of an external or internal file.

- OPEN statement effects

  Carriage control defaults to FORTRAN if the file is formatted, and the unit is connected to a terminal. Otherwise, carriage control defaults to LIST. The vms option affects the record length for direct access and relative organization files. The buffer size is increased by 1 to accommodate the deleted record character.

- Reading deleted records and ENDFILE records

The run-time direct access READ routine checks the first byte of the retrieved record. If this byte is '@' or NULL ("\0"), then an ATTACCNON error is returned. The run-time sequential access READ routine checks to see if the record it just read is one byte long and contains a Ctrl/Z. If this is true, it returns EOF.

Alternate Options

Linux and Mac OS X: None

Windows: `/Qvms`

See Also

-
- `align`
- `ccdefault`
- `check`

w

See keywords `none` *and* `nogeneral` *in warn*

W0, W1

See warn.

W0, W1

See warn.

Wa

Passes options to the assembler for processing.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Wa,option1[,option2,...]`

Windows:

`None`

Arguments

option	Is an assembler option. This option is not processed by the driver and is directly passed to the assembler.

Default

OFF	No options are passed to the assembler.

Description

This option passes one or more options to the assembler for processing. If the assembler is not invoked, these options are ignored.

Alternate Options

None

warn

Specifies diagnostic messages to be issued by the compiler.

IDE Equivalent

Windows: **General > Compile Time Diagnostics** (`/warn:all, /warn:none`)

Compilation Diagnostics > Treat Warnings as Errors (`/warn:[no]errors`)

Compilation Diagnostics > Treat Fortran Standard Warnings as Errors (`/warn:[no]stderrors`)

Compilation Diagnostics > Compile Time Diagnostics (`/warn:all, /warn:none`)

Compilation Diagnostics > Warn for Undeclared Symbols (`/warn:[no]declarations`)

Compilation Diagnostics > Warn for Unused Variables (`/warn:[no]unused`)

Compilation Diagnostics > Warn When Removing %LOC (`/warn:[no]ignore_loc`)

Compilation Diagnostics > Warn When Truncating Source Line (`/warn:[no]truncated_source`)

Compilation Diagnostics > Warn for Unaligned Data (`/warn:[no]alignments`)

Compilation Diagnostics > Warn for Uncalled Statement Function (`/warn:[no]uncalled`)

Compilation Diagnostics > Suppress Usage Messages (`/warn:[no]usage`)

Compilation Diagnostics > Check Routine Interfaces (`/warn:[no]interfaces`)

Linux: None

Mac OS X: **General > Compile Time Diagnostics** (`-warn all, -warn none`)

Compiler Diagnostics > Warn For Unaligned Data (`-warn [no]alignments`)

Compiler Diagnostics > Warn For Undeclared Symbols (`-warn [no]declarations`)

Compiler Diagnostics > Treat Warnings as Errors (`-warn error`)

Compiler Diagnostics > Warn When Removing %LOC (`-warn [no]ignore_loc`)

Compiler Diagnostics > Check Routine Interfaces (`-warn [no]interfaces`)

Compiler Diagnostics > Treat Fortran Standard Warnings As Errors (`-warn [no]stderrors`)

Compiler Diagnostics > Warn When Truncating Source Line (`-warn [no]truncated_source`)

Compiler Diagnostics > Warn For Uncalled Routine (`-warn [no]uncalled`)

Compiler Diagnostics > Warn For Unused Variables (`-warn [no]unused`)

Compiler Diagnostics > Suppress Usage Messages (`-warn [no]usage`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-warn [`*keyword*`]`

`-nowarn`

Windows:

`/warn[:keyword]`

`/nowarn`

Arguments

keyword Specifies the diagnostic messages to be issued. Possible values are:

`none`	Disables all warning messages.
`[no]alignments`	Determines whether warnings occur for data that is not naturally aligned.
`[no]declarations`	Determines whether warnings occur for any undeclared symbols.
`[no]errors`	Determines whether warnings are changed to errors.
`[no]general`	Determines whether warning messages and informational messages are issued by the compiler.
`[no]ignore_loc`	Determines whether warnings occur when `%LOC` is stripped from an actual argument.
`[no]interfaces`	Determines whether the compiler checks the interfaces of all SUBROUTINEs called and FUNCTIONs invoked in your compilation against an external set of interface blocks.
`[no]stderrors`	Determines whether warnings about Fortran standard violations are changed to errors.
`[no]truncated_source`	Determines whether warnings occur when source exceeds the maximum column width in fixed-format files.
`[no]uncalled`	Determines whether warnings occur when a statement function is never called
`[no]unused`	Determines whether warnings occur for declared variables that are never used.

[no]usage	Determines whether warnings occur for questionable programming practices.
all	Enables all warning messages except errors and stderrors.

Default

alignments	Warnings are issued about data that is not naturally aligned.
general	All information-level and warning-level messages are enabled.
usage	Warnings are issued for questionable programming practices.
nodeclarations	No errors are issued for undeclared symbols.
noerrors	Warning-level messages are not changed to error-level messages.
noignore_loc	No warnings are issued when %LOC is stripped from an argument.
nointerfaces	The compiler does not check interfaces of SUBROUTINEs called and FUNCTIONs invoked in your compilation against an external set of interface blocks.
nostderrors	Warning-level messages about Fortran standards violations are not changed to error-level messages.
notruncated_source	No warnings are issued when source exceeds the maximum column width in fixed-format files.
nouncalled	No warnings are issued when a statement function is not called.
nounused	No warnings are issued for variables that are declared but never used.

Description

This option specifies the diagnostic messages to be issued by the compiler.

Option	Description
warn none	Disables all warning messages. This is the same as specifying nowarn.
warn noalignments	Disables warnings about data that is not naturally aligned.
warn declarations	Enables error messages about any undeclared symbols. This option makes the default data type of a variable undefined (IMPLICIT NONE) rather than using the implicit Fortran rules.

Option	Description
warn errors	Tells the compiler to change all warning-level messages to error-level messages; this includes warnings about Fortran standards violations.
warn nogeneral	Disables all informational-level and warning-level diagnostic messages.
warn ignore_loc	Enables warnings when %LOC is stripped from an actual argument.
warn interfaces	Tells the compiler to check the interfaces of all SUBROUTINEs called and FUNCTIONs invoked in your compilation against a set of interface blocks stored separately from the source being compiled. The compiler generates a compile-time message if the interface used to invoke a routine does not match the interface defined in a .mod file external to the source (that is, in a .mod generated by option gen-interfaces as opposed to a .mod file USEd in the source). The compiler looks for these .mods in the current directory or in the directory specified by the include (-I) or -module option. warn interfaces turns on the option gen-interfaces by default. You can explicitly turn it off with /gen-interfaces- on Windows and -no-gen-interfaces on Linux and Mac OS X.
warn stderrors	Tells the compiler to change all warning-level messages about Fortran standards violations to error-level messages. This option sets the std03 option (Fortran 2003 standard). If you want Fortran 95 standards violations to become errors, you must specify options warn stderrors and std95.
warn truncated_source	Enables warnings when a source line exceeds the maximum column width in fixed-format source files. The maximum column width for fixed-format files is 72, 80, or 132, depending on the setting of the extend-source option. The warn truncated_source option has no effect on truncation; lines that exceed the maximum column width are always truncated. This option does not apply to free-format source files.
warn uncalled	Enables warnings when a statement function is never called.
warn unused	Enables warnings for variables that are declared but never used.

Option	Description
`warn nousage`	Disables warnings about questionable programming practices. Questionable programming practices, although allowed, often are the result of programming errors; for example: a continued character or Hollerith literal whose first part ends before the statement field and appears to end with trailing spaces. Note that the `/pad-source` option can prevent this error.
`warn all`	This is the same as specifying `warn`. This option does not set options `warn errors` or `warn stderrors`. To enable all the additional checking to be performed and force the severity of the diagnostic messages to be severe enough to not generate an object file, specify `warn all warn errors` or `warn all warn stderrors`. On Windows systems: In the Property Pages, **Custom** means that diagnostics will be specified on an individual basis.

Alternate Options

`warn none`	Linux and Mac OS X: `-nowarn`, `-w`, `-W0`, `-warn nogeneral` Windows: `/nowarn`,`/w`, `/W0`, `/warn:nogeneral`
`warn declarations`	Linux and Mac OS X: `-implicitnone`, `-u` Windows: `/4Yd`
`warn nodeclarations`	Linux and Mac OS X: None Windows: `/4Nd`
`warn general`	Linux and Mac OS X: `-W1` Windows: `/W1`
`warn nogeneral`	Linux and Mac OS X: `-W0`, `-w`, `-nowarn`, `-warn none` Windows: `/W0, /w, /nowarn, /warn:none`
`warn stderrors`	Linux and Mac OS X: `-e90`, `-e95`, `-e03` Windows: None
`warn nousage`	Linux and Mac OS X: `-cm` Windows: `/cm`
`warn all`	Linux and Mac OS X: `-warn` Windows: `/warn`

watch

Tells the compiler to display certain information to the console output window.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-watch [keyword]`

`-nowatch`

Windows:

`/watch[:keyword]`

`/nowatch`

Arguments

keyword Determines what information is displayed. Possible values are:

`none`	Disables `cmd` and `source`.
`[no]cmd`	Determines whether driver tool commands are displayed and executed.
`[no]source`	Determines whether the name of the file being compiled is displayed.
`all`	Enables `cmd` and `source`.

Default

`nowatch` Pass information and source file names are not displayed to the console output window.

Description

Tells the compiler to display processing information (pass information and source file names) to the console output window.

Option	Description
`watch none`	Tells the compiler to not display pass information and source file names to the console output window. This is the same as specifying `nowatch`.
`watch cmd`	Tells the compiler to display and execute driver tool commands.
`watch source`	Tells the compiler to display the name of the file being compiled.
`watch all`	Tells the compiler to display pass information and source file names to the console output window. This is the same as specifying `watch` with no *`keyword`*.

Alternate Options

`watch cmd`	Linux and Mac OS X: `-v` Windows: None

See Also

-
- `v`

WB

Turns a compile-time bounds check into a warning.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-WB`

Windows:

`/WB`

Arguments

None

Default

OFF	Compile-time bounds checks are errors.

Description

This option turns a compile-time bounds check into a warning.

Alternate Options

None

what

Tells the compiler to display its detailed version string.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-what`

Windows:

`/what`

Arguments

None

Default

OFF	The version strings are not displayed.

Description

This option tells the compiler to display its detailed version string.

Alternate Options

None

winapp

Tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.

IDE Equivalent

Windows: **Libraries > Use Common Windows Libraries**

Linux: None

Mac OS X: None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`None`

Windows:

`/winapp`

Arguments

None

Default

OFF	No graphics or Fortran Windows application is created.

Description

This option tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.

Alternate Options

Linux and Mac OS X: None

Windows: `/MG`

Winline

Enables diagnostics about what is inlined and what is not inlined.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-Winline
```

Windows:

```
None
```

Arguments

None

Default

OFF	No diagnostics are produced about what is inlined and what is not inlined.

Description

This option enables diagnostics about what is inlined and what is not inlined. The diagnostics depend on what interprocedural functionality is available.

Alternate Options

None

Wl

Passes options to the linker for processing.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Wl,option1[,option2,...]`

Windows:

`None`

Arguments

option	Is a linker option. This option is not processed by the driver and is directly passed to the linker.

Default

OFF	No options are passed to the linker.

Description

This option passes one or more options to the linker for processing. If the linker is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption,link,options`.

Alternate Options

None

See Also

-
- `Qoption`

Wp

Passes options to the preprocessor.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Wp,option1[,option2,...]`

Windows:

`None`

Arguments

option	Is a preprocessor option. This option is not processed by the driver and is directly passed to the preprocessor.

Default

OFF	No options are passed to the preprocessor.

Description

This option passes one or more options to the preprocessor. If the preprocessor is not invoked, these options are ignored.

This option is equivalent to specifying option `-Qoption, fpp, options`.

Alternate Options

None

See Also

-
- Qoption

x, Qx

Tells the compiler to generate optimized code specialized for the Intel processor that executes your program.

IDE Equivalent

Windows: **Code Generation > Intel Processor-Specific Optimization**

Optimization > Use Intel(R) Processor Extensions

Linux: None

Mac OS X: **Code Generation > Intel Processor-Specific Optimization**

Architectures

IA-32, Intel® 64 architectures

Syntax

Linux and Mac OS X:

-xprocessor

Windows:

/Qxprocessor

Arguments

processor	Indicates the processor for which code is generated. Many of the following descriptions refer to Intel® Streaming SIMD Extensions (Intel® SSE) and Supplemental Streaming SIMD Extensions (Intel® SSSE). Possible values are:

`Host`	Can generate instructions for the highest instruction set available on the compilation host processor. On Intel processors, this may correspond to the most suitable `-x` (Linux* and Mac OS* X) or `/Qx` (Windows*) option. On non-Intel processors, this may correspond to the most suitable `-m` (Linux and Mac OS X) or `/arch` (Windows) option. The resulting executable may not run on a processor different from the host in the following cases: • If the processor does not support all of the instructions supported by the host processor. • If the host is an Intel processor and the other processor is a non-Intel processor.
`AVX`	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).
`SSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
`SSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.

SSE3_ATOM	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology. Can generate MOVBE instructions, depending on the setting of option -minstruction (Linux and Mac OS) or /Qinstruction (Windows).
SSSE3	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. For Mac OS* X systems, this value is only supported on Intel® 64 architecture. This replaces value T, which is deprecated.
SSE3	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. For Mac OS* X systems, this value is only supported on IA-32 architecture.This replaces value P, which is deprecated.
SSE2	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. This value is not available on Mac OS* X systems. This replaces value N, which is deprecated.

Default

Windows* systems: None Linux* systems: None Mac OS* X systems using IA-32 architecture: SSE3 Mac OS* X systems using Intel® 64 architecture: SSSE3	On Windows systems, if neither /Qx nor /arch is specified, the default is /arch:SSE2. On Linux systems, if neither -x nor -m is specified, the default is -msse2.

Description

This option tells the compiler to generate optimized code specialized for the Intel processor that executes your program. It also enables optimizations in addition to Intel processor-specific optimizations. The specialized code generated by this option may run only on a subset of Intel processors.

This option can enable optimizations depending on the argument specified. For example, it may enable Intel® Streaming SIMD Extensions 4 (Intel® SSE4), Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Intel® Streaming SIMD Extensions 2 (Intel® SSE2), or Intel® Streaming SIMD Extensions (Intel® SSE) instructions.

The binaries produced by these values will run on Intel processors that support all of the features for the targeted processor. For example, binaries produced with SSE3 will run on an Intel® Core™ 2 Duo processor, because that processor completely supports all of the capabilities of the Intel® Pentium® 4 processor, which the SSE3 value targets. Specifying the SSSE3 value has the potential of using more features and optimizations available to the Intel® Core™ 2 Duo processor.

Do not use *processor* values to create binaries that will execute on a processor that is not compatible with the targeted processor. The resulting program may fail with an illegal instruction exception or display other unexpected behavior. For example, binaries produced with SSE3 may produce code that will not run on Intel® Pentium® III processors or earlier processors that do not support SSE3 instructions.

Compiling the main program with any of the *processor* values produces binaries that display a fatal run-time error if they are executed on unsupported processors. For more information, see *Optimizing Applications*.

If you specify more than one *processor* value, code is generated for only the highest-performing processor specified. The highest-performing to lowest-performing *processor* values are: SSE4.2, SSE4.1, SSSE3, SSE3, SSE2. Note that *processor* values AVX and SSE3_ATOM do not fit within this group.

Compiler options `m` and `arch` produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel processors.

Previous value O is deprecated and has been replaced by option `-msse3` (Linux and Mac OS X) and option `/arch:SSE3` (Windows).

Previous values W and K are deprecated. The details on replacements are as follows:

- Mac OS X systems: On these systems, there is no exact replacement for W or K. You can upgrade to the default option `-msse3` (IA-32 architecture) or option `-mssse3` (Intel® 64 architecture).

- Windows and Linux systems: The replacement for W is `-msse2` (Linux) or `/arch:SSE2` (Windows). There is no exact replacement for K. However, on Windows systems, `/QxK` is interpreted as `/arch:IA32`; on Linux systems, `-xK` is interpreted as `-mia32`. You can also do one of the following:
  - Upgrade to option `-msse2` (Linux) or option `/arch:SSE2` (Windows). This will produce one code path that is specialized for Intel® SSE2. It will not run on earlier processors
  - Specify the two option combination `-mia32 -axSSE2` (Linux) or `/arch:IA32 /QaxSSE2` (Windows). This combination will produce an executable that runs on any processor with IA-32 architecture but with an additional specialized Intel® SSE2 code path.

The `-x` and `/Qx` options enable additional optimizations not enabled with option `-m` or option `/arch`.

On Windows* systems, options `/Qx` and /arch are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning. Similarly, on Linux* and Mac OS* X systems, options `-x` and `-m` are mutually exclusive. If both are specified, the compiler uses the last one specified and generates a warning.

Alternate Options

None

See Also

-
-
- `ax`, `Qax`
- `m`
- `arch`
- `minstruction`, `Qinstruction`

X

Removes standard directories from the include file search path.

IDE Equivalent

Windows: **Preprocessor > Ignore Standard Include Path** (/noinclude)

Linux: None

Mac OS X: **Preprocessor > Ignore Standard Include Path** (/noinclude)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-X`

Windows:

`/X`

Arguments

None

Default

OFF	Standard directories are in the include file search path.

Description

This option removes standard directories from the include file search path. It prevents the compiler from searching the default path specified by the FPATH environment variable.

On Linux and Mac OS X systems, specifying `-X` (or `-noinclude`) prevents the compiler from searching in /usr/include for files specified in an INCLUDE statement.

You can use this option with the `I` option to prevent the compiler from searching the default path for include files and direct it to use an alternate path.

This option affects fpp preprocessor behavior and the USE statement.

Alternate Options

Linux and Mac OS X: `-nostdinc`

Windows: /noinclude

See Also

-
- `I`

Xlinker

Passes a linker option directly to the linker.

IDE Equivalent

None

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-Xlinker` *option*

Windows:

`None`

Arguments

option	Is a linker option.

Default

OFF	No options are passed directly to the linker.

Description

This option passes a linker option directly to the linker.
If `-Xlinker -shared` is specified, only `-shared` is passed to the linker and no special work is done to ensure proper linkage for generating a shared object. `-Xlinker` just takes whatever arguments are supplied and passes them directly to the linker.

If you want to pass compound options to the linker, for example "`-L $HOME/lib`", you must use the following method:

```
-Xlinker -L -Xlinker $HOME/lib
```

Alternate Options

None

See Also

-
- `shared`
- `link`

y

See syntax-only.

g, Zi, Z7

Tells the compiler to generate full debugging information in the object file.

IDE Equivalent

Windows: **General > Debug Information Format** (`/Z7, /Zd, /Zi`)

Linux: None

Mac OS X: **General > Generate Debug Information** (`-g`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-g`

Windows:

`/Zi`

`/Z7`

Arguments

None

Default

OFF	No debugging information is produced in the object file.

Description

This option tells the compiler to generate symbolic debugging information in the object file for use by debuggers.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off `O2` and makes `O0` (Linux and Mac OS X) or `Od` (Windows) the default unless `O2` (or another `O` option) is explicitly specified in the same command line.

On Linux systems using Intel® 64 architecture and Linux and Mac OS X systems using IA-32 architecture, specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option.

For more information on `Zi` and `Z7`, see keyword `full` in debug (Windows*).

Alternate Options

Linux and Mac OS X: None

Windows: `/debug:full` (or `/debug`)

See Also

-
-
-
- `Zd`

Zd

This option has been deprecated. Use keyword `minimal` *in debug (Windows*).*

zero, Qzero

Initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.

IDE Equivalent

Windows: **Data > Initialize Local Saved Scalars to Zero**

Linux: None

Mac OS X: **Data > Initialize Local Saved Scalars to Zero**

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

`-zero`

`-nozero`

Windows:

`/Qzero`

`/Qzero-`

Arguments

None

Default

`-nozero` or `/Qzero-`	Local scalar variables are not initialized to zero.

Description

This option initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.

Use `-save` (Linux and Mac OS X) or `/Qsave` (Windows) on the command line to make all local variables specifically marked as SAVE.

Alternate Options

None

See Also

-
-
- `save`

g, Zi, Z7

Tells the compiler to generate full debugging information in the object file.

IDE Equivalent

Windows: **General > Debug Information Format** (`/Z7`, `/Zd`, `/Zi`)

Linux: None

Mac OS X: **General > Generate Debug Information** (`-g`)

Architectures

IA-32, Intel® 64, IA-64 architectures

Syntax

Linux and Mac OS X:

```
-g
```

Windows:

```
/Zi
/Z7
```

Arguments

None

Default

OFF	No debugging information is produced in the object file.

Description

This option tells the compiler to generate symbolic debugging information in the object file for use by debuggers.

The compiler does not support the generation of debugging information in assemblable files. If you specify this option, the resulting object file will contain debugging information, but the assemblable file will not.

This option turns off `O2` and makes `O0` (Linux and Mac OS X) or `Od` (Windows) the default unless `O2` (or another `O` option) is explicitly specified in the same command line.

On Linux systems using Intel® 64 architecture and Linux and Mac OS X systems using IA-32 architecture, specifying the `-g` or `-O0` option sets the `-fno-omit-frame-pointer` option.

For more information on `Zi` and `Z7`, see keyword `full` in debug (Windows*).

Alternate Options

Linux and Mac OS X: None

Windows: `/debug:full` (or `/debug`)

See Also

-
-
-
- `Zd`

ZI

See keyword `none` *in libdir*

Zp

See keyword `recnbyte` *in align.*

Zs

See syntax-only.

Zx

Disables certain optimizations that make it difficult to debug optimized code.

IDE Equivalent

None

Architectures

IA-64 architecture

Syntax

Linux and Mac OS X:

`None`

Windows:

`/Zx`

Arguments

None

Default

OFF	Optimizations are not disabled.

Description

Disables certain optimizations, such as software pipelining and global scheduling, that make it difficult to debug resultant code because of speculation.

Alternate Options

None

Quick Reference Guides and Cross References

21

The topic summarizes Intel® Fortran compiler options used on Windows* OS, Linux* OS and Mac OS* X.

If you want to see the summarized Windows* OS options, see this topic.

If you want to see the summarized Linux* OS and Mac OS* X options, see this topic.

Windows* OS Quick Reference Guide and Cross Reference

The table in this section summarizes Intel® Fortran compiler options used on Windows* OS . Each summary also shows the equivalent compiler options on Linux* OS and Mac OS* X.

If you want to see the summarized Linux* OS and Mac OS* X options, see this table.

Some compiler options are only available on systems using certain architectures, as indicated by these labels:

Label	Meaning
i32	The option is available on systems using IA-32 architecture.
i64em	The option is available on systems using Intel® 64 architecture.
i64	The option is available on systems using IA-64 architecture.

If "only" appears in the label, the option is only available on the identified system or architecture.

If no label appears, the option is available on all supported systems and architectures.

For more details on the options, refer to the Alphabetical Compiler Options section.

The Intel® Fortran Compiler includes the Intel® Compiler Option Mapping tool. This tool lets you find equivalent options by specifying compiler option `-map-opts` (Linux OS and Mac OS X) or `/Qmap-opts` (Windows OS).

For information on conventions used in this table, see Conventions.

Quick Reference of Windows* OS Options

The following table summarizes all supported Windows* OS options. It also shows equivalent Linux* OS and Mac OS* X options, if any.

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/1	Executes at least one iteration of DO loops.	OFF	-1
/4I{2|4|8}	Specifies the default KIND for integer and logical variables; same as the /integer-size option.	/4I4	-i{2|4|8}
/4L{72|80|132}	Treats the statement field of each fixed-form source line as ending in column 72, 80, or 132; same as the /extend-source option.	/4L72	-72, -80, -132
/4Na, /4Ya	Determines where local variables are stored. /4Na is the same as /save. /4Ya is the same as /automatic.	/4Ya	None
/4Naltparam, /4Yaltparam	Determines whether alternate syntax is allowed for PARAMETER statements; same as the /altparam option).	/4Yaltparam	None
/4Nb, /4Yb	Determines whether checking is performed for	/4Nb	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	run-time failures (same as the /check option).		
/4Nd, /4Yd	Determines whether error messages are issued for undeclared symbols. /4Nd is the same as /warn:nodeclarations. /4Yd is the same as /warn:declarations.	/4Nd	-warn nodeclarations, -warn declarations
/4Nf, /4Yf	Specifies the format for source files. /4Nf is the same as /fixed. /4Yf is the same as /free.	/4Nf	-fixed, -free
/4Nportlib, /4Yportlib	Determines whether the compiler links to the library of portability routines.	/4Yportlib	None
/4Ns, /4Ys	Determines whether the compiler changes warning messages about Fortran standards violations to error messages. /4Ns is the same as /stand:none. /4Ys is the same as /stand:f90.	/4Ns	-stand none, -stand f90

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/4R8, /4R16	Specifies the default KIND for real and complex variables. /4R8 is the same as /real-size:64. /4R16 is the same as /real-size:128.	OFF	-real-size 64, -real-size 128
/align[:*keyword*]	Tells the compiler how to align certain data items.	keywords: nocommons nodcommons records nosequence	-align [*keyword*]
/allow:*keyword*	Determines how the fpp preprocessor treats Fortran end-of-line comments in preprocessor directive lines.	keyword: fpp_comments	/allow *keyword*
/[no]altparam	Allows alternate syntax (without parentheses) for PARAMETER statements.	/altparam	-[no]altparam
/arch:*processor* (i32, i64em)	Tells the compiler to generate optimized code specialized for the processor that executes your program; same as option /architecture.	varies; see option description	-arch *processor* or -m *processor* (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/asmattr:*keyword*	Specifies the contents of an assembly listing file.	/noasmattr	None
/asmfile[:*file* | *dir*]	Specifies that an assembly listing file should be generated.	/noasmfile	None
/assume:*keyword*	Tells the compiler to make certain assumptions.	keywords: [no]bscc [no]buffered_io [no]byterecl [no]cc_omp [no]dummy_aliases [no]ieee_fpe_flags [no]minus0 [no]old_boz [no]old_logical_ldio [no]old_maxminloc [no]old_unit_star [no]old_xor [no]protect_constants [no]protect_parens [no]realloc_lhs [no]source_include [no]std_mod_proc_name [no]underscore [no]writeable-strings	-assume *keyword*
/[no]automatic	Causes all variables to be allocated to the run-time stack; same as the /auto option.	/Qauto-scalar	-[no]automatic

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/bigobj	Increases the number of sections that an object file can contain.	OFF	None
/bintext:*string*	Places a text string into the object file (.obj) being generated by the compiler.	OFF	None
/c	Prevents linking.	OFF	-c
/C	Performs checking for all run-time failures; same as the /check:all option.	OFF	-C
/CB	Performs run-time checking on array subscript and character substring expressions; same as the /check:bounds option.	OFF	-CB
/ccdefault:*keyword*	Specifies the type of carriage control used when a file is displayed at a terminal screen.	/ccdefault:default	-ccdefault *keyword*
/check[:*keyword*]	Checks for certain conditions at run time.	/nocheck	-check [*keyword*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/cm	Disables all messages about questionable programming practices; same as specifying option /warn:nousage.	OFF	-cm
/compile-only	Causes the compiler to compile to an object file only and not link; same as the /c option.	OFF	None
/Qcomplex-limited-range[-]	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX.	/Qcomplex-limited-range-	-[no-]complex-limited-range
/convert:*keyword*	Specifies the format of unformatted files containing numeric data.	/convert:native	-convert *keyword*
/CU	Enables run-time checking for uninitialized variables. This option is the same as /check:uninit and /RTCu.	OFF	-CU

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/D*name* [=*value*]	Defines a symbol name that can be associated with an optional value.	/noD	-D*name* [=*value*]
/[no]d-lines	Compiles debugging statements indicated by the letter D in column 1 of the source code.	/nod-lines	-[no]d-lines
/[no]dbglibs	Tells the linker to search for unresolved references in a debug run-time library.	/nodbglibs	None
/debug:*keyword*	Specifies the type of debugging information generated by the compiler in the object file.	/debug:full (IDE) /debug:none (command line)	-debug *keyword* Note: the Linux* OS and Mac OS* X option takes different *keyword* s
/debug-parameters[:*keyword*]	Tells the compiler to generate debug information for PARAMETERs used in a program.	/nodebug-parameters	-debug-parameters [*keyword*]
/define:*name* [=*value*]	Defines a symbol name that can be associated with an optional value; same as the /D<name>[=value] option.	OFF	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/dll	Specifies that a program should be linked as a dynamic-link (DLL) library.	OFF	None
/double-size:*size*	Defines the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX variables.	/double-size:64	-double-size *size*
/E	Causes the Fortran preprocessor to send output to stdout.	OFF	-E
/EP	Causes the Fortran preprocessor to send output to stdout, omitting #line directives.	OFF	-EP
/error-limit:*n*	Specifies the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line.	/error-limit:30	-error-limit *n*
/exe:{*file* | *dir*}	Specifies the name for a built program or dynamic-link library.	OFF	-o

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/extend-source[:*size*]	Specifies the length of the statement field in a fixed-form source file.	/extend-source:72	-extend-source [*size*]
/extfor:ext	Specifies file extensions to be processed by the compiler as Fortran files.	OFF	None
/extfpp:ext	Specifies file extensions to be recognized as a file to be preprocessed by the Fortran preprocessor.	OFF	None
/extlnk:ext	Specifies file extensions to be passed directly to the linker.	OFF	None
/F*n*	Specifies the stack reserve amount for the program.	OFF	None
/f66	Tells the compiler to apply FORTRAN 66 semantics.	OFF	-f66
/f77rtl	Tells the compiler to use the run-time behavior of FORTRAN 77.	OFF	-f77rtl

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Fa[:*file* | *dir*]	Specifies that an assembly listing file should be generated; same as option /asmfile and /S.	OFF	-S
/FAc, /FAs, /FAcs	Specifies the contents of an assembly listing file. /FAc is the same as the /asmattr:machine option. /FAs is the same as the /asmattr:source option. /FAcs is the same as the /asmattr:all option.	OFF	None
/fast	Maximizes speed across the entire program.	OFF	-fast
/Fe*file*	Specifies the name for a built program or dynamic-link library; same as the /exe option.	OFF	-o
/FI	Specifies source files are in fixed format; same as the /fixed option.	determined by file suffix	-FI
/[no]fixed	Specifies source files are in fixed format.	determined by file suffix	-[no]fixed

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/[no]fltconsisten-cy	Enables improved floating-point consistency.	/nofltconsistency	-[no]fltconsisten-cy
/Fm[*file*]	Tells the linker to generate a link map file; same as the /map option.	OFF	None
/Fo*file*	Specifies the name for an object file; same as the /object option.	OFF	None
/fp:*keyword*	Controls the semantics of floating-point calculations.	/fp:fast=1	-fp-model *keyword*
/[no]fpconstant	Tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.	/nofpconstant	-[no]fpconstant
/fpe:*n*	Specifies the floating-point exception handling level for the main program at run-time.	/fpe:3	-fpe*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/fpe-all:*n*	Specifies the floating-point exception handling level for each routine at run-time.	/fpe-all:3	-fpe-all=*n*
/fpp [*n*] /fpp[="*option*"]	Runs the Fortran preprocessor on source files before compilation.	/nofpp	- fpp[*n*] - fpp[="*option*"]
/fpscomp[:*keyword*]	Specifies compatibility with Microsoft* Fortran PowerStation or Intel® Fortran.	/fpscomp:libs	-fpscomp [*keyword*]
/FR	Specifies source files are in free format; same as the /free option.	determined by file suffix	-FR
/[no]free	Specifies source files are in free format.	determined by file suffix	-[no]free
/G2 (i64 only)	Optimizes application performance for systems using IA-64 architecture.	OFF	None
/G2-p9000 (i64 only)	Optimizes for Dual-Core Intel® Itanium® 2 Processor 9000 series.	ON	-mtune itanium2-p9000

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/G{5|6|7} (i32, i64em)	Optimizes application performance for systems using IA-32 architecture and Intel® 64 architecture. These options have been deprecated.	/G7	None
/GB	Optimizes for Intel® Pentium® Pro, Pentium® II and Pentium® III processors; same as the /G6 option.	OFF	None
/Ge	Enables stack-checking for all functions. Deprecated.	OFF	None
/gen-interfaces[:[no]source]	Tells the compiler to generate an interface block for each routine in a source file.	/nogen-interfaces	-gen-interfaces [[no]source]
/Gm	Tells the compiler to use calling convention CVF; same as the /iface:cvf option.	OFF	None
/Gs[n]	Disables stack-checking for routines with a specified number of	/Gs4096	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	bytes of local variables and compiler temporaries.		
/GS[-]	Determines whether the compiler generates code that detects some buffer overruns.	/GS-	-f[no-]stack-security-check
/Gz	Tells the compiler to use calling convention STDCALL; same as the /iface:stdcall option.	OFF	None
/heap-arrays[:*size*]	Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.	/heap-arrays-	-heap-arrays [*size*]
/help[*category*]	Displays all available compiler options or a category of compiler options; same as the /? option.	OFF	-help
/homeparams	Tells the compiler to store parameters passed in registers to the stack.	OFF	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/hotpatch (i32, i64em)	Tells the compiler to prepare a routine for hotpatching.	OFF	None
/I:*dir*	Specifies a directory to add to the include path.	OFF	-I*dir*
/iface:*keyword*	Specifies the default calling convention and argument-passing convention for an application.	/iface:default	None
/include	Specifies a directory to add to the include path; same as the /I option.	OFF	None
/inline[:*keyword*]	Specifies the level of inline function expansion.	OFF	None
/[no]intconstant	Tells the compiler to use FORTRAN 77 semantics to determine the kind parameter for integer constants.	/nointconstant	-[no]intconstant
/integer-size:*size*	Specifies the default KIND for integer and logical variables.	/integer-size:32	-integer-size *size*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
`/LD`	Specifies that a program should be linked as a dynamic-link (DLL) library.	OFF	None
`/libdir`[`:`*`keyword`*]	Controls whether linker options for search libraries are included in object files generated by the compiler.	`/libdir:all`	None
`/libs:`*`keyword`*	Tells the linker to search for unresolved references in a specific run-time library.	`/libs:static`	None
`/link`	Passes options to the linker at compile time.	OFF	None
`/[no]logo`	Displays the compiler version information.	`/logo`	`-[no]logo`
`/map[:`*`file`*`]`	Tells the linker to generate a link map file.	`/nomap`	None
`/MD` and `/MDd`	Tells the linker to search for unresolved references in a multithreaded, debug, dynamic-link run-time library.	OFF	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/MDs and /MDsd	Tells the linker to search for unresolved references in a single-threaded, dynamic-link run-time library.	OFF	None
/MG	Tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.	OFF	None
/ML and /MLd	Tells the linker to search for unresolved references in a single-threaded, static run-time library.	i32, i64: /ML i64em: OFF	None
/module:*path*	Specifies the directory where module files should be placed when created and where they should be searched for.	OFF	-module *path*
/MP[:*n*]	Creates multiple processes that can be used to compile large numbers of source files at the same time.	OFF	-multiple-process-es[:*n*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/MT and /MTd	Tells the linker to search for unresolved references in a multithreaded, static run-time library.	i32, i64: OFF i64em: /MT/noreen-trancy	None
/MW	Tells the linker to search for unresolved references in a Fortran QuickWin library; same as /libs:qwin.	OFF	None
/MWs	Tells the linker to search for unresolved references in a Fortran standard graphics library; same as /libs:qwins.	OFF	None
/names:*keyword*	Specifies how source code identifiers and external names are interpreted.	/names:uppercase	-names *keyword*
/nbs	Tells the compiler to treat the backslash character (\) as a normal character in character literals; same as the /assume:nobscc option.	/nbs	-nbs

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/noinclude	Removes standard directories from the include file search path; same as the /X option.	OFF	None
/O[*n*]	Specifies the code optimization for applications.	/O2	-O[*n*]
/Ob*n*	Specifies the level of inline function expansion. *n* = 0, 1, or 2.	/Ob2 if /O2 is in effect or /O3 is specified /Ob0 if /Od is specified	-inline-level=*n*
/object:*file*	Specifies the name for an object file.	OFF	None
/Od	Disables optimizations.	OFF	-O0
/Og	Enables global optimizations.	/Og	None
/Op	Enables improved floating-point consistency.	OFF	-mp
/optimize:*n*	Affects optimizations performed by the compiler; *n* = 1, 2, 3, or 4.	/optimize:3 or /optimize:4	-O*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Os	Enables optimizations that do not increase code size and produces smaller code size than O2.	OFF (unless /O1 is specified)	-Os
/Ot	Enables all speed optimizations.	/Ot (unless /O1 is specified)	None
/Ox	Same as the /O2 option.	/Ox	-O2
/Oy[-] (i32 only)	Determines whether EBP is used as a general-purpose register in optimizations.	/Oy (unless /Od is specified)	-f[no-]omit-frame-pointer (i32, i64em)
/P	Causes the Fortran preprocessor to send output to a file, which is named by default; same as the -preprocess-only option.	OFF	-P
/[no]pad-source	Specifies padding for fixed-form source records.	/nopad-source	-[no]pad-source
/pdbfile[:*file*]	Specifies that any debug information generated by the compiler should be saved to a program database file.	/nopdbfile	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/preprocess-only	Causes the Fortran preprocessor to send output to a file.	/nopreprocess-only	-preprocess-only
/Qansi-alias	Tells the compiler to assume the program adheres to the Fortran Standard type aliasability rules.	/Qansi-alias	-ansi-alias
/Qauto	Causes all variables to be allocated on the stack, rather than in local static storage.	/Qauto-scalar	-auto
/Qauto-scalar	Causes allocation of scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL to the run-time stack.	/Qauto-scalar	-auto-scalar
/Qautodouble	Makes default real and complex variables 8 bytes long; same as the /real-size:64 option.	OFF	-autodouble
/Qax*p* (i32, i64em)	Tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel	OFF	-ax*p* (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	processors if there is a performance benefit.		
/Qchkstk[-] (i64 only)	Enables stack probing when the stack is dynamically expanded at run-time.	/Qchkstk	None
/Qcommon-args	Tells the compiler that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with COMMON variables that are assigned; same as /assume:dummy_aliases.	OFF	-common-args
/Qcomplex-limited-range[-]	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX.	/Qcomplex-limited-range-	-[no-]complex-limited-range
/Qcpp	Runs the Fortran preprocessor on source files before compilation; same as the /fpp option.	OFF	-cpp

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qd-lines[-]	Compiles debugging statements indicated by the letter D in column 1 of the source code; can also be specified as /d-lines.	OFF	-[no]d-lines
/Qdiag-*type*: *diag-list*	Controls the display of diagnostic information.	OFF	-diag-*type* *diag-list*
/Qdiag-dump	Tells the compiler to print all enabled diagnostic messages and stop compilation.	OFF	-diag-dump
/Qdiag-enable:sc-include (i32, i64em)	Tells the Source Checker to analyze include files and source files when issuing diagnostic message. This is equivalent to deprecated option /Qdiag-enable:sv-include.	OFF	-diag-enable sc-include (i32, i64em)
/Qdiag-enable:sc-parallel (i32, i64em)	Enables analysis of parallelization in source code (parallel lint diagnostics).	OFF	-diag-enable sc-parallel (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qdiag-error-limit:*n*	Specifies the maximum number of errors allowed before compilation stops.	/Qdiag-error-limit:30	-diag-error-limit*n*
/Qdiag-file[:*file*]	Causes the results of diagnostic analysis to be output to a file.	OFF	-diag-file[= *file*]
/Qdiag-file-append[:*file*]	Causes the results of diagnostic analysis to be appended to a file.	OFF	-diag-file-append[= *file*]
/Qdiag-id-numbers[-]	Tells the compiler to display diagnostic messages by using their ID number values.	/Qdiag-id-numbers	-[no-]diag-id-numbers
/Qdiag-once:*id*[,*id*,...]	Tells the compiler to issue one or more diagnostic messages only once	OFF	-diag-once *id*[,*id*,...]
/Qdps	Specifies that the alternate syntax for PARAMETER statements is allowed; same as the /altparam option.	/Qdps	-dps
/Qdyncom "common1,common2,..."	Enables dynamic allocation of common blocks at run time.	OFF	-dyncom "common1,common2,..."

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qextend-source	This is the same as specifying option /extend-source:132.	OFF	-extend-source *size*
/Qfast-transcendentals[-]	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.	/Qfast-transcendentals	-[no-]fast-transcendentals
/Qfma[-] (i64 only)	Enables the combining of floating-point multiplies and add/subtract operations.	/Qfma	-[no-]fma (i64 only; Linux* OS only)
/Qfnalign[:*n*] (i32, i64em)	Tells the compiler to align functions on an optimal byte boundary.	/Qfnalign-	-falign-functions[=*n*] (i32, i64em)
/Qfnsplit[-] (i32, i64)	Enables function splitting.	/Qfnsplit-	-[no-]fnsplit (i64 only; Linux* OS only)
/Qfp-port[-] (i32, i64em)	Rounds floating-point results after floating-point operations.	/Qfp-port-	-[no-]fp-port (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
`/Qfp-relaxed[-]` (i64 only)	Enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt.	`/Qfp-relaxed-`	`-[no-]fp-relaxed` (i64 only; Linux* OS only)
`/Qfp-specula-tion:`*mode*	Tells the compiler the mode in which to speculate on floating-point operations.	`/Qfp-specula-tion:fast`	`-fp-specula-tion=`*mode*
`/Qfp-stack-check` (i32, i64em)	Generates extra code after every function call to ensure that the FP (floating-point) stack is in the expected state.	OFF	`-fp-stack-check` (i32, i64em)
`/Qfpp[`*n*`]`	Runs the Fortran preprocessor on source files before compilation.	`/nofpp`	`-fpp[`*n*`]`
`/Qftz[-]`	Flushes denormal results to zero.	i64: `/Qftz-` i32, i64em: `/Qftz`	`-[no-]ftz`
`/Qglobal-hoist[-]`	Enables certain optimizations that can move memory loads to a point earlier in the	`/Qglobal-hoist-`	`-[no-]global-hoist`

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	program execution than where they appear in the source.		
/QIA64-fr32 (i64 only)	Disables use of high floating-point registers.	OFF	None
/QIfist (i32 only)	Enables fast float-to-integer conversions; same as option /Qrcd.	OFF	None
/Qimsl	Tells the compiler to link to the IMSL* Fortran Numerical Library* (IMSL* library).	OFF	None
/Qinline-debug-info	Produces enhanced source position information for inlined code.	OFF	-inline-debug-info
/Qinline-dl-limport[-]	Determines whether dllimport functions are inlined.	/Qinline-dl-limport	None
/Qinline-factor==*n*	Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.	/Qinline-factor-	-inline-factor=*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
`/Qinline-forcein-line`	Specifies that an inline routine should be inlined whenever the compiler can do so.	OFF	`-inline-forcein-line`
`/Qinline-max-per-compile=`*n*	Specifies the maximum number of times inlining may be applied to an entire compilation unit.	`/Qinline-max-per-compile-`	`-inline-max-per-compile=`*n*
`/Qinline-max-per-routine=`*n*	Specifies the maximum number of times the inliner may inline into a particular routine.	`/Qinline-max-per-routine-`	`-inline-max-per-routine=`*n*
`/Qinline-max-size=`*n*	Specifies the lower limit for the size of what the inliner considers to be a large routine.	`/Qinline-max-size-`	`-inline-max-size=`*n*
`/Qinline-max-to-tal-size=`*n*	Specifies how much larger a routine can normally grow when inline expansion is performed.	`/Qinline-max-to-tal-size-`	`-inline-max-to-tal-size=`*n*
`/Qinline-min-size=`*n*	Specifies the upper limit for the size of what the inliner considers to be a small routine.	`/Qinline-min-size-`	`-inline-min-size=`*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qinstruction:[no]movbe (i32, i64em)	Determines whether MOVBE instructions are generated for Intel processors.	OFF	-minstruction=[no]movbe (i32, i64em)
/Qinstrument-functions[-]	Determines whether function entry and exit points are instrumented.	/Qinstrument-functions-	-f[no-]instrument-functions
/Qip[-]	Enables additional single-file interprocedural optimizations.	OFF	-[no-]ip
/Qip-no-inlining	Disables full and partial inlining enabled by -ip.	OFF	-ip-no-inlining
/Qip-no-pinlining (i32, i64em)	Disables partial inlining.	OFF	-ip-no-pinlining (i32, i64em)
/QIPF-flt-eval-method0 (i64 only)	Tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program. Deprecated.	OFF	-[no-]IPF-flt-eval-method0 (i64 only; Linux* OS only)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/QIPF-fltacc[-] (i64 only)	Tells the compiler to apply optimizations that affect floating-point accuracy. Deprecated.	/QIPF-fltacc-	-IPF-fltacc (i64 only; Linux* OS only)
/QIPF-fma (i64 only)	Enables the combining of floating-point multiplies and add/subtract operations. Deprecated; use /Qfma.	/QIPF-fma	-IPF-fma (i64 only; Linux* OS only)
/QIPF-fp-relaxed (i64 only)	Enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt. Deprecated; use /Qfp-relaxed.	/QIPF-fp-relaxed-	-IPF-fp-relaxed (i64 only; Linux* OS only)
/Qipo[n]	Enables multifile IP optimizations between files.	OFF	-ipo[n]
/Qipo-c	Generates a multifile object file that can be used in further link steps.	OFF	-ipo-c

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qipo-jobs: *n*	Specifies the number of commands to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).	/Qipo-jobs:1	-ipo-jobs*n*
/Qipo-S	Generates a multifile assembly file that can be used in further link steps.	OFF	-ipo-S
/Qipo-separate	Generates one object file per source file.	OFF	-ipo-separate (Linux* OS only)
/Qivdep-parallel (i64 only)	Tells the compiler that there is no loop-carried memory dependency in any loop following an IVDEP directive.	OFF	-ivdep-parallel (i64 only; Linux* OS only)
/Qkeep-static-consts[-]	Tells the compiler to preserve allocation of variables that are not referenced in the source.	/Qkeep-static-consts-	-f[no-]keep-static-consts
/Qlocation,*string*,*dir*	Specifies a directory as the location of the specified tool in *string* .	OFF	-Qlocation,*string*,*dir*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qlowercase	Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase; same as the /names:lowercase option.	OFF	-lowercase
/Qmap-opts	Maps one or more Windows* compiler options to their equivalent on a Linux* OS system (or vice versa).	OFF	-map-opts
/Qmkl[=*lib*	Tells the compiler to link to certain parts of the Intel® Math Kernel Library.	OFF	-mkl[=*lib*
/Qnobss-init	Places any variables that are explicitly initialized with zeros in the DATA section.	OFF	-no-bss-init
/Qonetrip	This is the same as specifying option /onetrip.	OFF	-onetrip
/Qopenmp	Enables the parallelizer to generate	OFF	-openmp

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	multithreaded code based on OpenMP* directives.		
/Qopenmp-lib:*type*	Lets you specify an OpenMP* run-time library to use for linking.	/Qopenmp-lib:legacy	-openmp-lib *type* (Linux* OS only)
/Qopenmp-link:*library*	Lets you specify an OpenMP* run-time library to use for linking.	/Qopenmp-lib:legacy	-openmp-link *library* (Linux* OS only)
/Qopenmp-profile	Enables analysis of OpenMP* applications.	OFF	-openmp-profile (Linux* OS only)
/Qopenmp-report[*n*]	Controls the OpenMP parallelizer's level of diagnostic messages.	/Qopenmp-report1	-openmp-report[*n*]
/Qopenmp-stubs	Enables compilation of OpenMP programs in sequential mode.	OFF	-openmp-stubs
/Qopenmp-threadprivate:*type*	Lets you specify an OpenMP* threadprivate implementation.	/Qopenmp-threadprivate:legacy	-openmp-threadprivate *type* (Linux* OS only)
/Qopt-block-factor:*n*	Lets you specify a loop blocking factor.	OFF	-opt-block-factor=*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
`/Qopt-jump-tables:`*keyword*	Enables or disables generation of jump tables for switch statements.	`/Qopt-jump-tables:default`	`-opt-jump-tables=`*keyword*
`/Qopt-loadpair[-]` (i64 only)	Enables or disables loadpair optimization.	`/Qopt-loadpair-`	`-[no-]opt-load-pair` (i64 only; Linux* OS only)
`/Qopt-mem-band-width`*n* (i64 only)	Enables performance tuning and heuristics that control memory bandwidth use among processors.	`/Qopt-mem-band-width0` for serial compilation; `/Qopt-mem-bandwidth1` for parallel compilation	`-opt-mem-band-width`*n* (i64 only; Linux* OS only)
`/Qopt-mod-version-ing[-]` (i64 only)	Enables or disables versioning of modulo operations for certain types of operands.	`/Qopt-mod-version-ing-`	`-[no-]opt-mod-versioning` (i64 only; Linux* OS only)
`/Qopt-multi-ver-sion-aggres-sive[-]` (i32, i64em)	Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.	`/Qopt-multi-ver-sion-aggressive-`	`-[no-]opt-multi-version-aggres-sive` (i32, i64em)
`/Qopt-prefetch`[:*n*]	Enables prefetch insertion optimization.	i64: `/Qopt-prefetch` i32, i64em: `/Qopt-prefetch-`	`-opt-prefetch`[=*n*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qopt-prefetch-initial-values[-] (i64 only)	Enables or disables prefetches that are issued before a loop is entered.	/Qopt-prefetch-initial-values	-[no-]opt-prefetch-initial-values (i64 only; Linux* OS only)
/Qopt-prefetch-issue-excl-hint[-] (i64 only)	Determines whether the compiler issues prefetches for stores with exclusive hint.	/Qopt-prefetch-issue-excl-hint-	-[no-]opt-prefetch-issue-excl-hint (i64 only; Linux* OS only)
/Qopt-prefetch-next-iteration[-][:*n*] (i64 only)	Enables or disables prefetches for a memory access in the next iteration of a loop.	-opt-prefetch-next-iteration	/Qopt-prefetch-next-iteration (i64 only; Linux* OS only)
/Qopt-ra-region-strategy[:*keyword*] (i32, i64em)	Selects the method that the register allocator uses to partition each routine into regions.	/Qopt-ra-region-strategy:default	-opt-ra-region-strategy[=*keyword*] (i32, i64em)
/Qopt-report:*n*	Tells the compiler to generate an optimization report to *stderr*.	/Qopt-report:2	-opt-report *n*
/Qopt-report-file:*file*	Tells the compiler to generate an optimization report named *file*.	OFF	-opt-report-file=*file*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qopt-report-help	Displays the optimizer phases available for report generation.	OFF	-opt-report-help
/Qopt-report-phase:*phase*	Specifies an optimizer phase to use when optimization reports are generated.	OFF	-opt-report-phase=*phase*
/Qopt-report-routine:*string*	Generates a report on all routines or the routines containing the specified *string*.	OFF	-opt-report-routine=*string*
/Qopt-streaming-stores:*keyword* (i32, i64em)	Enables generation of streaming stores for optimization.	/Qopt-streaming-stores:auto	-opt-streaming-stores *keyword* (i32, i64em)
/Qopt-subscript-in-range[-] (i32, i64em)	Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.	/Qopt-subscript-in-range-	-[no-]opt-subscript-in-range (i32, i64em)
/Qoption,*string*,*options*	Passes *options* to the tool specified in *string*.	OFF	-Qoption,*string*,*options*
/Qpad	Enables the changing of the variable and array memory layout.	/Qpad-	-pad

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qpad-source	This is the same as specifying option /pad-source.	/Qpad-source-	-pad-source
/Qpar-adjust-stack:*n* (i32, i64em)	Tells the compiler to generate code to adjust the stack size for a fiber-based main thread.	/Qpar-adjust-stack:0	None
/Qpar-affinity:[*modifier*,...] *type*[,*permute*] [,*offset*]	Specifies thread affinity.	OFF	-par-affinity=[*modifier*,...]*type*[,*permute*][,*offset*] (Linux* OS only)
/Qpar-num-threads:*n*	Specifies the number of threads to use in a parallel region.	OFF	-par-num-threads=*n*
/Qpar-report[*n*]	Controls the diagnostic information reported by the auto-parallelizer.	/Qpar-report1	-par-report[*n*]
/Qpar-runtime-control[-]	Generates code to perform run-time checks for loops that have symbolic loop bounds.	/Qpar-runtime-control-	-[no-]par-run-time-control
/Qpar-schedule-*keyword* [[:] *n*]	Specifies a scheduling algorithm for DO loop iterations.	OFF	-par-schedule-*keyword*[=*n*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qpar-thresh-old[[:]*n*]	Sets a threshold for the auto-parallelization of loops.	/Qpar-thresh-old:100	-par-threshold[*n*]
/Qparallel	Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.	OFF	-parallel
/Qpc*n* (i32, i64em)	Enables control of floating-point significand precision.	/Qpc64	-pc*n* (i32, i64em)
/Qprec	Improves floating-point precision and consistency.	OFF	-mp1
/Qprec-div[-]	Improves precision of floating-point divides.	/Qprec-div-	-[no-]prec-div
/Qprec-sqrt (i32, i64em)	Improves precision of square root implementations.	/Qprec-sqrt-	-prec-sqrt (i32, i64em)
/Qprefetch	Enables prefetch insertion optimization. Deprecated; use /Qopt-prefetch	i64: /Qprefetch i32, i64em: /Qprefetch-	-prefetch

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qprof-data-order[-]	Enables or disables data ordering if profiling information is enabled.	/Qprof-data-order-	-[no-]prof-data-order (Linux* OS only)
/Qprof-dir *dir*	Specifies a directory for profiling information output files.	OFF	-prof-dir *dir*
/Qprof-file *file*	Specifies a file name for the profiling summary file.	OFF	-prof-file *file*
/Qprof-func-order[-]	Enables or disables function ordering if profiling information is enabled.	/Qprof-func-order-	-[no-]prof-func-order (Linux* OS only)
/Qprof-gen[:*keyword*]	Produces an instrumented object file that can be used in profile-guided optimization.	/Qprof-gen-	-prof-gen[=*keyword*]
/Qprof-genx	Produces an instrumented object file that includes extra source position information. Deprecated; use /Qprof-gen:srcpos.	OFF	-prof-genx

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qprof-hotness-threshold:*n*	Lets you set the hotness threshold for function grouping and function ordering.	OFF	-prof-hotness-threshold=*n* (Linux* OS only)
/Qprof-src-dir[-]	Determines whether directory information of the source file under compilation is considered when looking up profile data records.	/Qprof-src-dir	-[no-]prof-src-dir
/Qprof-src-root:*dir*	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.	OFF	-prof-src-root=*dir*
/Qprof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.	OFF	-prof-src-root-cwd
/Qprof-use[:*arg*]	Enables the use of profiling information during optimization.	OFF	-prof-use[=*arg*]
/Qrcd (i32, i64em)	Enables fast float-to-integer conversions.	OFF	-rcd (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qrct (i32, i64em)	Sets the internal FPU rounding control to Truncate.	OFF	-rct (i32, i64em)
/Qsafe-cray-ptr	Tells the compiler that Cray* pointers do not alias other variables.	OFF	-safe-cray-ptr
/Qsave	Causes variables to be placed in static memory.	/Qauto-scalar	-save
/Qsave-temps[-]	Tells the compiler to save intermediate files created during compilation.	.obj files are saved	-[no-]save-temps
/Qscalar-rep[-] (i32 only)	Enables scalar replacement performed during loop transformation (requires /O3).	/Qscalar-rep-	-[no-]scalar-rep (i32 only)
/Qsfalign[n] (i32 only)	Specifies stack alignment for functions. *n* is 8 or 16.	/Qsfalign8	None
/Qsox[-]	Tells the compiler to save the compilation options and version number in the Windows* OS object file.	/Qsox-	-[no-]sox (Linux OS only)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
`/Qtcheck`	Enables analysis of threaded applications.	OFF	`-tcheck` (Linux* OS only)
`/Qtcollect`[: *lib*]	Inserts instrumentation probes calling the Intel(R) Trace Collector API.	OFF	`-tcollect`[= *lib*] (Linux* OS only)
`/Qtcollect-filter`[=*file*]	Lets you enable or disable the instrumentation of specified functions.	OFF	`-tcollect-filter` [*file*] (Linux* OS only)
`/Qtprofile`	Generates instrumentation to analyze multi-threading performance.	OFF	`-tprofile` (Linux* OS only)
`/Qtrapuv`	Initializes stack local variables to an unusual value.	OFF	`-ftrapuv`
`/Qunroll`[:*n*]	Tells the compiler the maximum number of times to unroll loops; same as the `/unroll[:n]` option.	`/Qunroll`	`-unroll`[=*n*]
`/Qunroll-aggressive[-]` (i32, i64em)	Determines whether the compiler uses more aggressive unrolling for certain loops.	`/Qunroll-aggressive-`	`-[no-]unroll-aggressive` (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Quppercase	Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase; same as the /names:uppercase option.	/Quppercase	-uppercase
/Quse-asm[-] (i64 only)	Tells the compiler to produce objects through the assembler.	/Quse-asm-	-[no-]use-asm
/Quse-msasm-symbols (i32,i64em)	Tells the compiler to use a dollar sign ("$") when producing symbol names.	OFF	None
/Quse-vcdebug (i32 only)	Tells the compiler to issue debug information compatible with the Visual C++ debugger.	OFF	None
/Qvc7.1 /Qvc8 /Qvc9 (i32, i64em)	Specifies compatibility with Microsoft* Visual C++ or Microsoft* Visual Studio.	varies; see option description	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qvec[-] (i32, i64em)	Enables or disables vectorization and transformations enabled for vectorization.	/Qvec	-[no-]vec (i32, i64em)
/Qvec-guard-write[-] (i32, i64em)	Tells the compiler to perform a conditional check in a vectorized loop.	/Qvec-guard-write-	-[no-]vec-guard-write (i32, i64em)
/Qvec-report[*n*] (i32, i64em)	Controls the diagnostic information reported by the vectorizer.	/Qvec-report1	-vec-report[*n*] (i32, i64em)
/Qvec-thresh-old[[:]*n*] (i32, i64em)	Sets a threshold for the vectorization of loops.	/Qvec-thresh-old100	-vec-threshold[*n*] (i32, i64em)
/Qvms	Causes the run-time system to behave like HP Fortran for OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*) in certain ways; same as the /vms option.	/novms	-vms
/Qx*processor* (i32, i64em)	Tells the compiler to generate optimized code specialized for the Intel processor that executes your program.	varies; see the option description	-x*processor* (i32, i64em)

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/Qzero[-]	Initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.	OFF	-[no]zero
/real-size:*size*	Specifies the default KIND for real variables.	/real-size:32	-real-size *size*
/[no]recursive	Tells the compiler that all routines should be compiled for possible recursive execution.	/norecursive	-[no]recursive
/reentrancy:*keyword*	Tells the compiler to generate reentrant code to support a multithreaded application.	/noreentrancy	-reentrancy *keyword*
/RTCu	Enables run-time checking for uninitialized variables; same as option /check:uninit.	OFF	-check uninit or -CU
/S	Causes the compiler to compile to an assembly file only and not link.	OFF	-S

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/source:*file*	Tells the compiler to compile the file as a Fortran source file.	OFF	None
/stand:*keyword*	Tells the compiler to issue compile-time messages for nonstandard language elements.	/nostand	-stand *keyword*
/static	Prevents linking with shared libraries.	/static	-static (Linux* OS only)
/syntax-only	Tells the compiler to check only for correct syntax.	OFF	-syntax-only
/Tf *file*	Tells the compiler to compile the file as a Fortran source file; same as the /source option.	OFF	-Tf *file*
/[no]threads	Tells the linker to search for unresolved references in a multithreaded run-time library.	i32, i64: /nothreads i64em: /threads	-[no]threads
/[no]traceback	Tells the compiler to generate extra information in the object file to provide source file traceback	/notraceback	-[no]traceback

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	information when a severe error occurs at run time.		
/tune:*keyword* (i32, i64em)	Determines the version of the architecture for which the compiler generates instructions.	/tune:pn4	-tune *keyword* (i32, i64em)
/u	Undefines all previously defined preprocessor values.	OFF	None Note: the Linux* OS and Mac OS* X option -u is not the same
/U*name*	Undefines any definition currently in effect for the specified symbol; same as the /undefine option.	OFF	-U*name*
/undefine:*name*	Undefines any definition currently in effect for the specified symbol; same as option /U.	OFF	None
/unroll[:*n*]	Tells the compiler the maximum number of times to unroll loops. This is the same as /Qunroll, which is the recommended option to use.	/unroll	-unroll[=*n*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/us	Tells the compiler to append an underscore character to external user-defined names; same as the /assume:underscore option.	OFF	-us
/V*string*	Places the text string specified into the object file (.obj) being generated by the compiler; same as the /bintext option.	OFF	None
/[no]vms	Causes the run-time system to behave like HP* Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).	/novms	-[no]vms
/w	Disables all warning messages; same as specifying option /warn:none or /warn:nogeneral.	OFF	-w
/W*n*	Disables (*n*=0) or enables (*n*=1) all warning messages.	/W1	-W*n*

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/warn:*keyword*	Specifies diagnostic messages to be issued by the compiler.	keywords: alignments general usage nodeclarations noerrors noignore_loc nointerfaces nostderrors notruncated_source nouncalled nounused	-warn *keyword*
/watch[:*keyword*]	Tells the compiler to display certain information to the console output window.	/nowatch	-watch [*keyword*]
/WB	Turns a compile-time bounds check error into a warning.	OFF	-WB
/what	Tells the compiler to display its detailed version string.	OFF	-what
/winapp	Tells the compiler to create a graphics or Fortran Windows application and link against the most commonly used libraries.	OFF	None

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
/X	Removes standard directories from the include file search path.	OFF	-X
/Zd	Tells the compiler to generate line numbers and minimal debugging information. Deprecated; use option /debug:minimal.	OFF	None
/Zi or /Z7	Tells the compiler to generate full debugging information in the object file; same as the /debug:full or /debug option.	OFF	-g
/Zl	Prevents any linker search options from being included into the object file; same as the /libdir:none or /nolibdir option.	OFF	None
/Zp[*n*]	Aligns fields of records and components of derived types on the smaller of the size boundary specified or	/Zp16	-Zp[*n*]

Option	Description	Default	Equivalent Option on Linux* OS and Mac OS* X
	the boundary that will naturally align them; same as the /align:rec*n byte* option.		
/Zs	Tells the compiler to perform syntax checking only; same as the /syntax-only option.	OFF	None

See Also

- Quick Reference Guides and Cross References
- -map-opts, /Qmap-opts

Linux* OS and Mac OS* X Quick Reference Guide and Cross Reference

The table in this section summarizes Intel® Fortran compiler options used on Linux* OS and Mac OS* X. Each summary also shows the equivalent compiler options on Windows* OS.

If you want to see the summarized Windows* OS options, see this table.

Some compiler options are only available on systems using certain architectures, as indicated by these labels:

Label	Meaning
i32	The option is available on systems using IA-32 architecture.
i64em	The option is available on systems using Intel® 64 architecture.
i64	The option is available on systems using IA-64 architecture.

If "only" appears in the label, the option is only available on the identified system or architecture.

If no label appears, the option is available on all supported systems and architectures.

For more details on the options, refer to the Alphabetical Compiler Options section.

The Intel® Fortran Compiler includes the Intel® Compiler Option Mapping tool. This tool lets you find equivalent options by specifying compiler option `-map-opts` (Linux OS and Mac OS X) or `/Qmap-opts` (Windows OS).

For information on conventions used in this table, see Conventions.

Quick Reference of Linux OS and Mac OS X Options

The following table summarizes all supported Linux OS and Mac OS X options. It also shows equivalent Windows OS options, if any.

Option	Description	Default	Equivalent Option on Windows* OS
`-1`	Executes at least one iteration of DO loops.	OFF	`/1`
`-66`	Tells the compiler to use FORTRAN 66 semantics.	OFF	None
`-72`, `-80`, `-132`	Treats the statement field of each fixed-form source line as ending in column 72, 80, or 132; same as the `-extend-source` option.	`-72`	`/4L{72|80|132}`
`-align [`*keyword*`]`	Tells the compiler how to align certain data items.	keywords: `nocommons` `nodcommons` `records` `nosequence`	`/align[:`*keyword*`]`
`-allow [no]fpp_comments`	Determines how the fpp preprocessor treats Fortran end-of-line	`-allow fpp_comments`	`/allow:[no]fpp_comments`

Option	Description	Default	Equivalent Option on Windows* OS
	comments in preprocessor directive lines.		
-[no]altparam	Allows alternate syntax (without parentheses) for PARAMETER statements.	-altparam	/[no]altparam
-[no-]ansi-alias	Tells the compiler to assume the program adheres to the Fortran Standard type aliasability rules.	-ansi-alias	/Qansi-alias[-]
-arch *keyword* (i32, i64em)	Tells the compiler to generate optimized code specialized for the processor that executes your program.	varies; see the option description	/arch:*keyword* (i32, i64em)

Option	Description	Default	Equivalent Option on Windows* OS

`-assume` *keyword*	Tells the compiler to make certain assumptions.	keywords:
		`[no]bscc`
		`[no]buffered_io`
		`[no]byterecl`
		`[no]cc_omp`
		`[no]dummy_aliases`
		`[no]ieee_fpe_flags`
		`[no]minus0`
		`[no]old_boz`
		`[no]old_logical_ldio`
		`[no]old_maxminloc`
		`[no]old_unit_star`
		`[no]old_xor`
		`[no]protect_constants`
		`[no]protect_parens`
		`[no]realloc_lhs`
		`[no]source_include`
		`[no]std_mod_proc_name`
		`[no]underscore`
		`[no]2underscores`
		`[no]writeable-strings`
			`/assume:`*keyword*

`-[no]automatic`	Causes all variables to be allocated to the run-time stack; same as the `-auto` option.	`-auto-scalar`	`/[no]automatic`
`-auto-scalar`	Causes allocation of scalar variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL to the run-time stack.	`-auto-scalar`	`/Qauto-scalar`

Option	Description	Default	Equivalent Option on Windows* OS
`-autodouble`	Makes default real and complex variables 8 bytes long; same as the `-real-size 64` option.	OFF	`/Qautodouble`
`-ax`*`processor`* (i32, i64em)	Tells the compiler to generate multiple, processor-specific auto-dispatch code paths for Intel processors if there is a performance benefit.	OFF	`/Qax`*`processor`* (i32, i64em)
`-B`*`dir`*	Specifies a directory that can be used to find include files, libraries, and executables.	OFF	None
`-Bdynamic` (Linux OS only)	Enables dynamic linking of libraries at run time.	OFF	None
`-Bstatic` (Linux OS only)	Enables static linking of a user's library.	OFF	None
`-c`	Causes the compiler to compile to an object file only and not link.	OFF	`/c`
`-CB`	Performs run-time checks on whether array subscript and	OFF	`/CB`

Option	Description	Default	Equivalent Option on Windows* OS
	substring references are within declared bounds; same as the `-check bounds` option.		
`-ccdefault` *keyword*	Specifies the type of carriage control used when a file is displayed at a terminal screen.	`-ccdefault de-fault`	`/ccde-fault:`*keyword*
`-check` [*keyword*]	Checks for certain conditions at run time.	`-nocheck`	`/check`[`:`*keyword*]
`-cm`	Disables all messages about questionable programming practices; same as specifying option `-warn nousage`.	OFF	`/cm`
`-common-args`	Tells the compiler that dummy (formal) arguments to procedures share memory locations with other dummy arguments or with COMMON variables that are assigned. This option is the same as `-assume dummy_aliases`.	OFF	`/Qcommon-args`

Option	Description	Default	Equivalent Option on Windows* OS
`-[no-]complex-limited-range`	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX.	`-no-complex-limited-range`	`/Qcomplex-limited-range[-]`
`-convert` *keyword*	Specifies the format of unformatted files containing numeric data.	`-convert native`	`/convert:`*keyword*
`-cpp`	Runs the Fortran preprocessor on source files prior to compilation.	OFF	`/Qcpp`
`-CU`	Enables run-time checking for uninitialized variables. This option is the same as `-check uninit`.	OFF	`/CU`
`-cxxlib[=`*dir* `]`	Tells the compiler to link using the C++ run-time libraries provided by gcc.	`-no-cxxlib`	None
`-cxxlib-nostd`	Prevents the compiler from linking with the standard C++ library.	`-no-cxxlib`	None

Option	Description	Default	Equivalent Option on Windows* OS
`-D`*`name`* [=*`value`*]	Defines a symbol name that can be associated with an optional value.	`-noD`	`/D`*`name`* [=*`value`*]
`-[no]d-lines`	Compiles debugging statements indicated by the letter D in column 1 of the source code.	`-nod-lines`	`/[no]d-lines`or `/Qd-lines`
`-DD`	Compiles debugging statements indicated by the letter D in column 1 of the source code; same as the `-d-lines` option.	`-nod-lines`	`/d-lines`
`-debug` *`keyword`*	Specifies settings that enhance debugging.	`-debug none`	`/debug:`*`keyword`* Note: the Windows option takes different *`keyword`* s
`-debug-parameters` [*`keyword`*]	Tells the compiler to generate debug information for PARAMETERs used in a program.	`-nodebug-parame-ters`	`/debug-parame-ters`[:*`keyword`*]
`-diag-`*`type diag-list`*	Controls the display of diagnostic information.	OFF	`/Qdiag-`*`type`* : *`diag-list`*
`-diag-dump`	Tells the compiler to print all enabled diagnostic messages and stop compilation.	OFF	`/Qdiag-dump`

Option	Description	Default	Equivalent Option on Windows* OS
`-diag-enable sc-include` (i32, i64em)	Tells the Source Checker to analyze include files and source files when issuing diagnostic message. This option is equivalent to `-diag-enable sv-include`.	OFF	`/Qdiag-enable:sc-include` (i32, i64em)
`-diag-enable sc-parallel` (i32, i64em)	Enables analysis of parallelization in source code (parallel lint diagnostics).	OFF	`/Qdiag-enable:sc-parallel` (i32, i64em)
`-diag-error-limit` *n*	Specifies the maximum number of errors allowed before compilation stops.	`-diag-error-limit 30`	`/Qdiag-error-limit:`*n*
`-diag-file`[= *file*]	Causes the results of diagnostic analysis to be output to a file.	OFF	`/Qdiag-file[:`*file*`]`
`-diag-file-append`[= *file*]	Causes the results of diagnostic analysis to be appended to a file.	OFF	`/Qdiag-file-append[:`*file* `]`
`-[no-]diag-id-numbers`	Tells the compiler to display diagnostic messages by using their ID number values.	`-diag-id-numbers`	`/Qdiag-id-numbers[-]`

Option	Description	Default	Equivalent Option on Windows* OS
`-diag-once` *id*[,*id*,...]	Tells the compiler to issue one or more diagnostic messages only once	OFF	`/Qdiag-once:`*id*[,*id*,...]
`-double-size` *size*	Defines the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX variables.	`-double-size 64`	`/double-size:`*size*
`-dps`	Specifies that the alternate syntax for PARAMETER statements is allowed; same as the `-altparam` option.	`-dps`	`/Qdps`
`-dryrun`	Specifies that driver tool commands should be shown but not executed.	OFF	None
`-dumpmachine`	Displays the target machine and operating system configuration.	OFF	None
`-dynamic-linker`*file* (Linux OS only)	Specifies a dynamic linker in *file* other than the default.	OFF	None
`-dynamiclib` (i32, i64em; Mac OS X only)	Invokes the *libtool* command to generate dynamic libraries.	OFF	None

Option	Description	Default	Equivalent Option on Windows* OS
`-dyncom "com-mon1,common2,..."`	Enables dynamic allocation of common blocks at run time.	OFF	`/Qdyncom "com-mon1,common2,..."`
`-E`	Causes the Fortran preprocessor to send output to stdout.	OFF	`/E`
`-e03, -e95, -e90`	Causes the compiler to issue errors instead of warnings for nonstandard Fortran; same as the `-warn stderrors` option.	OFF	None
`-EP`	Causes the Fortran preprocessor to send output to stdout, omitting #line directives.	OFF	`/EP`
`-error-limit` *n*	Specifies the maximum number of error-level or fatal-level compiler errors allowed for a file specified on the command line; same as `-diag-error-limit`.	`-error-limit 30`	`/error-limit:`*n*
`-extend-source` *size*	Specifies the length of the statement field in a fixed-form source file.	`-extend-source 72`	`/extend-source:`*size*

Option	Description	Default	Equivalent Option on Windows* OS
`-f66`	Tells the compiler to use FORTRAN 66 semantics.	OFF	`/f66`
`-[no]f77rtl`	Tells the compiler to use FORTRAN 77 run-time behavior.	OFF	`/[no]f77rtl`
`-f[no-]alias`	Determines whether aliasing should be assumed in the program.	ON	None
`-falign-functions`[=*n*] (i32, i64em)	Tells the compiler to align functions on an optimal byte boundary.	`-no-falign-functions`	`/Qfnalign[:`*n*`]` (i32, i64em)
`-falign-stack`[=*mode*] (i32 only)	Tells the compiler to align functions on an optimal byte boundary.	`-falign-stack=default`	None
`-fast`	Maximizes speed across the entire program.	OFF	`/fast`
`-[no-]fast-transcendentals`	Enables the compiler to replace calls to transcendental functions with faster but less precise implementations.	ON	`/Qfast-transcendentals[-]`
`-fcode-asm`	Produces an assembly file with optional machine code annotations.	OFF	`/FAc`

Option	Description	Default	Equivalent Option on Windows* OS
`-f[no-]exceptions`	Enables exception handling table generation.	`-fno-exceptions`	None
`-f[no-]fnalias`	Specifies that aliasing should be assumed within functions.	OFF	`-ffnalias`
`-FI`	Specifies source files are in fixed format; same as the `-fixed` option.	determined by file suffix	`/FI`
`-f[no-]inline`	Tells the compiler to inline functions declared with cDEC$ ATTRIBUTES FORCEINLINE.	`-fno-inline`	None
`-f[no-]inline-functions`	Enables function inlining for single file compilation.	`-finline-functions`	`/Ob2`
`-finline-limit=`*n*	Lets you specify the maximum size of a function to be inlined.	OFF	None
`-f[no-]instrument-functions`	Determines whether function entry and exit points are instrumented.	`-fno-instrument-functions`	`/Qinstrument-functions[-]`
`-[no]fixed`	Specifies source files are in fixed format.	determined by file suffix	`/[no]fixed`

Option	Description	Default	Equivalent Option on Windows* OS
`-f[no-]keep-static-consts`	Tells the compiler to preserve allocation of variables that are not referenced in the source.	`-fno-keep-static-consts`	`/Qkeep-static-consts[-]`
`-[no]fltconsistency`	Enables improved floating-point consistency.	OFF	`/[no]fltconsistency`
`-[no-]fma` (i64 only; Linux OS only)	Enables the combining of floating-point multiplies and add/subtract operations.	`-fma`	`/Qfma[-]` (i64 only)
`-f[no-]math-errno`	Tells the compiler that *errno* can be reliably tested after calls to standard math library functions.	`-fno-math-errno`	None
`-fminshared`	Tells the compiler to treat a compilation unit as a component of a main program and not to link it as a shareable object.	OFF	None
`-[no-]fnsplit` (i64 only; Linux OS only)	Enables function splitting.	OFF	`/Qfnsplit[-]` (i32, i64)

Option	Description	Default	Equivalent Option on Windows* OS
`-f[no-]omit-frame-pointer` (i32, i64em)	Determines whether EBP is used as a general-purpose register in optimizations. This is the same as specifying option `-fp`, which is deprecated.	`-fomit-frame-pointer` (unless option `-O0` or `-g` is specified)	`/Oy[-]` (i32 only)
`-fp-model` *keyword*	Controls the semantics of floating-point calculations.	`-fp-model fast`	`/fp:`*keyword*
`-[no-]fp-port`	Rounds floating-point results after floating-point operations, so rounding to user-declared precision happens at assignments and type conversions (some impact on speed).	`-no-fp-port`	`/Qfp-port[-]`
`-[no-]fp-relaxed` (i64 only; Linux OS only)	Enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt.	`-no-fp-relaxed`	`/Qfp-relaxed[-]` (i64 only)

Option	Description	Default	Equivalent Option on Windows* OS
`-fp-specula-tion=`*mode*	Tells the compiler the mode in which to speculate on floating-point operations.	`-fp-specula-tion=fast`	`/Qfp-specula-tion:`*mode*
`-fp-stack-check` (i32, i64em)	Generates extra code after every function call to ensure that the FP (floating-point) stack is in the expected state.	OFF	`/Qfp-stack-check` (i32, i64em)
`-[no]fpconstant`	Tells the compiler that single-precision constants assigned to double-precision variables should be evaluated in double precision.	OFF	`/[no]fpconstant`
`-fpe`*n*	Specifies the floating-point exception handling level for the main program at run-time.	`-fpe3`	`/fpe:`*n*
`-fpe-all=`*n*	Specifies the floating-point exception handling level for each routine at run-time.	`-fpe-all3`	`/fpe-all:`*n*
`-f[no-]pic`, `-fPIC`	Generates position-independent code.	`-fno-pic`	None

Option	Description	Default	Equivalent Option on Windows* OS
`-fpie` (Linux OS only)	Tells the compiler to generate position-independent code.	OFF	None
`-fpp[`*`n`*`]` or `-fpp[="`*`option`*`"]`	Runs the Fortran preprocessor on source files prior to compilation.	`-nofpp`	`/fpp[`*`n`*`]` or `/fpp[="`*`option`*`"]`
`-fpscomp` [*`keyword`*]	Specifies compatibility with Microsoft* Fortran PowerStation or Intel® Fortran.	`-fpscomp libs`	`/fpscomp[:`*`keyword`*`]`
`-FR`	Specifies source files are in free format; same as the `-free` option.	determined by file suffix	`/FR`
`-fr32` (i64 only; Linux OS only)	Disables use of high floating-point registers.	OFF	None
`-[no]free`	Specifies source files are in free format.	determined by file suffix	`/[no]free`
`-fsource-asm`	Produces an assembly file with optional source code annotations.	OFF	`/FAs`
`-f[no-]stack-security-check`	Determines whether the compiler generates code that detects some buffer	`-fno-stack-security-check`	`/GS[-]`

Option	Description	Default	Equivalent Option on Windows* OS
	overruns; same as `-f[no-]stack-protector`.		
`-fsyntax-only`	Specifies that the source file should be checked only for correct syntax; same as `-syntax-only`.	OFF	None
`-ftrapuv`	Initializes stack local variables to an unusual value.	OFF	`/Qtrapuv`
`-[no-]ftz`	Flushes denormal results to zero.	i64: `-no-ftz` i32, i64em: `-ftz`	`/Qftz[-]`
`-[no-]func-groups` (i32, i64em; Linux OS only)	Enables or disables function grouping if profiling information is enabled. This option is deprecated, use `-prof-func-groups`.	`-no-func-groups`	None
`-funroll-loops`	Tells the compiler to unroll user loops based on the default optimization heuristics; same as `-unroll`, which is the recommended option.	`-funroll-loops`	`/Qunroll`

Option	Description	Default	Equivalent Option on Windows* OS
`-fverbose-asm`	Produces an assembly file with compiler comments, including options and version information.	`-fno-verbose-asm`	None
`-fvisibility=`*keyword* `-fvisibility-`*keyword*`=` *file*	Specifies the default visibility for global symbols; the 2nd form indicates symbols in a file.	`-fvisibility=default`	None
`-g`	Produces symbolic debug information in the object file.	OFF	`/Zi, /Z7`
`-gdwarf2`	Enables generation of debug information using the DWARF2 format.	OFF	None
`-gen-interfaces` [*`[no]source`*]	Tells the compiler to generate an interface block for each routine in a source file.	`-nogen-interfaces`	`/gen-interfaces`[:*`[no]source`*]
`-[no-]global-hoist`	Enables certain optimizations that can move memory loads to a point earlier in the program execution than where they appear in the source.	`-global-hoist`	`/Qglobal-hoist[-]`

Option	Description	Default	Equivalent Option on Windows* OS
`-heap-arrays [size]`	Puts automatic arrays and arrays created for temporary computations on the heap instead of the stack.	`-no-heap-arrays`	`/heap-arrays[:size]`
`-help [category]`	Displays the list of compiler options.	OFF	`/help [category]`
`-Idir`	Specifies a directory to add to the include path.	OFF	`/Idir`
`-i-dynamic`	Links Intel-provided libraries dynamically. This is a deprecated option; use `-shared-intel`.	OFF	None
`-i-static`	Links Intel-provided libraries statically. This is a deprecated option; use `-static-intel`.	OFF	None
`-i{2|4|8}`	Specifies the default KIND for integer and logical variables; same as the `-integer-size` option.	`-i4`	`/4I{2|4|8}`
`-idirafterdir`	Adds a directory to the second include file search path.	OFF	None

Option	Description	Default	Equivalent Option on Windows* OS
`-implicitnone`	Sets the default type of a variable to undefined; same as option `warn declarations`.	OFF	`/4Yd`
`-inline-debug-info`	Produces enhanced source position information for inlined code.	OFF	`/Qinline-debug-info`
`-inline-factor=`*n*	Specifies the percentage multiplier that should be applied to all inlining options that define upper limits.	`-no-inline-factor`	`/Qinline-factor=`*n*
`-inline-forceinline`	Specifies that an inline routine should be inlined whenever the compiler can do so.	OFF	`/Qinline-forceinline`
`-inline-level=`*n*	Specifies the level of inline function expansion. *n* = 0, 1, or 2.	`-inline-level=2` if `-O2` is in effect `-inline-level=0` if `-O0` is specified	`/Ob`*n*
`-inline-max-per-compile=`*n*	Specifies the maximum number of times inlining may be applied to an entire compilation unit.	`-no-inline-max-per-compile`	`/Qinline-max-per-compile=`*n*

Option	Description	Default	Equivalent Option on Windows* OS
`-inline-max-per-routine=`*n*	Specifies the maximum number of times the inliner may inline into a particular routine.	`-no-inline-max-per-routine`	`/Qinline-max-per-routine=`*n*
`-inline-max-size=`*n*	Specifies the lower limit for the size of what the inliner considers to be a large routine.	`-no-inline-max-size`	`/Qinline-max-size=`*n*
`-inline-max-total-size=`*n*	Specifies how much larger a routine can normally grow when inline expansion is performed.	`-no-inline-max-total-size`	`/Qinline-max-total-size=`*n*
`-inline-min-size=`*n*	Specifies the upper limit for the size of what the inliner considers to be a small routine.	`-no-inline-min-size`	`/Qinline-min-size=`*n*
`-[no]intconstant`	Tells the compiler to use FORTRAN 77 semantics to determine the KIND for integer constants.	`-nointconstant`	`/[no]intconstant`
`-integer-size` *size*	Specifies the default KIND for integer and logical variables.	`-integer-size 32`	`/integer-size:`*size*

Option	Description	Default	Equivalent Option on Windows* OS
`-[no-]ip`	Enables additional single-file interprocedural optimizations.	OFF	`/Qip[-]`
`-ip-no-inlining`	Disables full and partial inlining enabled by `-ip`.	OFF	`/Qip-no-inlining`
`-ip-no-pinlining`	Disables partial inlining.	OFF	`/Qip-no-pinlining`
`-IPF-flt-eval-method0` (i64 only; Linux OS only)	Tells the compiler to evaluate the expressions involving floating-point operands in the precision indicated by the variable types declared in the program. Deprecated.	OFF	`/QIPF-flt-eval-method0` (i64 only)
`-IPF-fltacc` (i64 only; Linux OS only)	Tells the compiler to apply optimizations that affect floating-point accuracy. Deprecated.	`-no-IPF-fltacc`	`/QIPF-fltacc` (i64 only)
`-IPF-fma` (i64 only; Linux OS only)	Enables the combining of floating-point multiplies and add/subtract	`-IPF-fma`	`/QIPF-fma` (i64 only)

Option	Description	Default	Equivalent Option on Windows* OS
	operations. Deprecated; use `-fma`.		
`-IPF-fp-relaxed` (i64 only; Linux OS only)	Enables use of faster but slightly less accurate code sequences for math functions, such as divide and sqrt. Deprecated; use `-fp-relaxed`.	`-no-IPF-fp-relaxed`	`/QIPF-fp-relaxed` (i64 only)
`-ipo`[*n*]	Enables multifile IP optimizations between files.	OFF	`/Qipo`[*n*]
`-ipo-c`	Generates a multifile object file that can be used in further link steps.	OFF	`/Qipo-c`
`-ipo-jobs`*n*	Specifies the number of commands to be executed simultaneously during the link phase of Interprocedural Optimization (IPO).	`-ipo-jobs1`	`/Qipo-jobs:`*n*
`-ipo-S`	Generates a multifile assembly file that can be used in further link steps.	OFF	`/Qipo-S`
`-ipo-separate` (Linux OS only)	Generates one object file per source file.	OFF	`/Qipo-separate`

Option	Description	Default	Equivalent Option on Windows* OS
-isystem*dir*	Specifies a directory to add to the start of the system include path.	OFF	None
-ivdep-parallel (i64 only; Linux OS only)	Tells the compiler that there is no loop-carried memory dependency in any loop following an IVDEP directive.	OFF	/Qivdep-parallel (i64 only)
-l*string*	Tells the linker to search for a specified library when linking.	OFF	None
-L*dir*	Tells the linker where to search for libraries before searching the standard directories.	OFF	None
-[no]logo	Displays compiler version information.	-nologo	/[no]logo
-lowercase	Causes the compiler to ignore case differences in identifiers and to convert external names to lowercase; same as the -names lowercase option.	-lowercase	/Qlowercase
-m[*processor*] (i32, i64em)	Tells the compiler to generate optimized code specialized for	varies; see option description	/arch

Option	Description	Default	Equivalent Option on Windows* OS
	the processor that executes your program.		
`-m32, -m64` (i32, i64em)	Tells the compiler to generate code for IA-32 architecture or Intel® 64 architecture, respectively.	OFF	None
`-map-opts` (Linux OS only)	Maps one or more Linux* OS compiler options to their equivalent on a Windows* system (or vice versa).	OFF	`/Qmap-opts`
`-march=`*`processor`* (i32, i64em; Linux OS only)	Tells the compiler to generate code for a specified processor.	i32: OFF i64em: `-march=pentium4`	None
`-mcmodel=`*`mem_model`* (i64em only; Linux OS only)	Tells the compiler to use a specific memory model to generate code and store data.	`-mcmodel=small`	None
`-mdynamic-no-pic` (i32 only; Mac OS X only)	Generates code that is not position-independent but has position-independent external references.	OFF	None
`-mieee-fp`	Tells the compiler to use IEEE floating point comparisons.	OFF	`/fltconsistency`

Option	Description	Default	Equivalent Option on Windows* OS
	This is the same as specifying option `-fltconsistency` or `-mp`.		
`-minstruc-tion=[no]movbe` (i32, i64em)	Determines whether MOVBE instructions are generated for Intel processors.	OFF	/Qinstruction:[no]movbe (i32, i64em)
`-mixed_str_len_arg`	Tells the compiler that the hidden length passed for a character argument is to be placed immediately after its corresponding character argument in the argument list.	OFF	`/iface:mixed_str_len_arg`
`-mkl[=`*lib*	Tells the compiler to link to certain parts of the Intel® Math Kernel Library.	OFF	`/Qmkl[=`*lib*
`-module` *path*	Specifies the directory where module files should be placed when created and where they should be searched for.	OFF	`/module:`*path*
`-mp`	Enables improved floating-point consistency.	OFF	`/Op`

Option	Description	Default	Equivalent Option on Windows* OS
`-mp1`	Improves floating-point precision and consistency.	OFF	`/Qprec`
`-m[no-]relax` (i64 only)	Determines whether the compiler passes linker option `-relax` to the linker.	`-mno-relax`	None
`-mtune=`*`processor`*	Performs optimizations for a particular processor. `-mtune=itanium2` is equivalent to `/G2`.	i32: `-mtune=pentium4` i64: `-mtune=itanium2-p9000`	None
`-multiple-processes=` *`n`*	Creates multiple processes that can be used to compile large numbers of source files at the same time.	OFF	`/MP:` *`n`*
`-names` *`keyword`*	Specifies how source code identifiers and external names are interpreted.	`-names lowercase`	`/names:`*`keyword`*
`-nbs`	Tells the compiler to treat the backslash character (\) as a normal character in character literals; same as the `-assume nobscc` option.	`-nbs`	`/nbs`

Option	Description	Default	Equivalent Option on Windows* OS
-no-bss-init	Tells the compiler to place in the DATA section any variables explicitly initialized with zeros.	OFF	/Qno-bss-init
-nodefaultlibs	Prevents the compiler from using standard libraries when linking.	OFF	None
-nodefine	Specifies that all preprocessor definitions apply only to fpp and not to Intel® Fortran conditional compilation directives.	OFF	/nodefine
-nofor-main	Specifies the main program is not written in Fortran, and prevents the compiler from linking for_main.o into applications.	OFF	None
-noinclude	Prevents the compiler from searching in a directory previously added to the include path for files specified in an INCLUDE statement.	OFF	/noinclude

Option	Description	Default	Equivalent Option on Windows* OS
`-nolib-inline`	Disables inline expansion of standard library or intrinsic functions.	OFF	None
`-nostartfiles`	Prevents the compiler from using standard startup files when linking.	OFF	None
`-nostdinc`	Removes standard directories from the include file search path; same as the `-X` option.	OFF	None
`-nostdlib`	Prevents the compiler from using standard libraries and startup files when linking.	OFF	None
`-nus`	Disables appending an underscore to external user-defined names; same as the `-assume nounder-score` option.	OFF	None
`-o`*`file`*	Specifies the name for an output file.	OFF	None
`-O`[*n*]	Specifies the code optimization for applications.	`-O2`	`/O`[*n*]
`-O0`	Disables all optimizations.	OFF	`/Od`

Option	Description	Default	Equivalent Option on Windows* OS
-onetrip	Executes at least one iteration of DO loops.	OFF	/onetrip
-openmp	Enables the parallelizer to generate multithreaded code based on OpenMP* directives.	OFF	/Qopenmp
-openmp-lib *type* (Linux OS only)	Lets you specify an OpenMP* run-time library to use for linking.	-openmp-lib legacy	/Qopenmp-lib:*type*
-openmp-link *library*	Controls whether the compiler links to static or dynamic OpenMP run-time libraries.	-openmp-link dynamic	/Qopenmp-link:*library*
-openmp-profile (Linux OS only)	Enables analysis of OpenMP* applications.	OFF	/Qopenmp-profile
-openmp-report[*n*]	Controls the OpenMP parallelizer's level of diagnostic messages.	-openmp-report1	/Qopenmp-report[*n*]
-openmp-stubs	Enables compilation of OpenMP programs in sequential mode.	OFF	/Qopenmp-stubs
-openmp-threadprivate *type* (Linux OS only)	Lets you specify an OpenMP* threadprivate implementation.	-openmp-threadprivate legacy	/Qopenmp-threadprivate:*type*

Option	Description	Default	Equivalent Option on Windows* OS
`-opt-block-factor=`*n*	Lets you specify a loop blocking factor.	OFF	`/Qopt-block-factor:`*n*
`-opt-jump-tables=`*keyword*	Enables or disables generation of jump tables for switch statements.	`-opt-jump-tables=default`	`/Qopt-jump-tables:`*keyword*
`-[no-]opt-loadpair` (i64 only; Linux OS only)	Enables or disables loadpair optimization.	`-no-opt-loadpair`	`/Qopt-loadpair[-]` (i64 only)
`-opt-malloc-options=`*n* (i32, i64em)	Lets you specify an alternate algorithm for malloc().	`-opt-malloc-options=0`	None
`-opt-mem-bandwidth`*n* (i64 only; Linux OS only)	Enables performance tuning and heuristics that control memory bandwidth use among processors.	`-opt-mem-bandwidth0` for serial compilation; `-opt-mem-bandwidth1` for parallel compilation	`/Qopt-mem-bandwidth`*n* (i64 only)
`-[no-]opt-mod-versioning` (i64 only; Linux OS only)	Enables or disables versioning of modulo operations for certain types of operands.	`-no-opt-mod-versioning`	`/Qopt-mod-versioning[-]` (i64 only)
`-[no-]opt-multi-version-aggressive` (i32, i64em)	Tells the compiler to use aggressive multi-versioning to check for pointer aliasing and scalar replacement.	`-no-opt-multi-version-aggressive`	`/Qopt-multi-version-aggressive[-]` (i32, i64em)

Option	Description	Default	Equivalent Option on Windows* OS
`-opt-prefetch`[=*n*]	Enables prefetch insertion optimization.	i64: `-opt-prefetch` i32, i64em: `-no-opt-prefetch`	`/Qopt-prefetch`[:*n*]
`-[no-]opt-prefetch-initial-values` (i64 only; Linux OS only)	Enables or disables prefetches that are issued before a loop is entered.	`-opt-prefetch-initial-values`	`/Qopt-prefetch-initial-values[-]` (i64 only)
`-[no-]opt-prefetch-issue-excl-hint` (i64 only; Linux OS only)	Determines whether the compiler issues prefetches for stores with exclusive hint.	`-no-opt-prefetch-issue-excl-hint`	`/Qopt-prefetch-issue-excl-hint[-]` (i64 only)
`-[no-]opt-prefetch-next-iteration` (i64 only; Linux OS only)	Enables or disables prefetches for a memory access in the next iteration of a loop.	`-opt-prefetch-next-iteration`	`/Qopt-prefetch-next-iteration[-]`[:*n*] (i64 only)
`-opt-ra-region-strategy`[=*keyword*] (i32, i64em)	Selects the method that the register allocator uses to partition each routine into regions.	`-opt-ra-region-strategy=default`	`/Qopt-ra-region-strategy`[:*keyword*] (i32, i64em)
`-opt-report` [*n*]	Tells the compiler to generate an optimization report to stderr.	`-opt-report 2`	`/Qopt-report[:`*n*]
`-opt-report-file=`*file*	Specifies the name for an optimization report.	OFF	`/Qopt-report-file:`*file*

Option	Description	Default	Equivalent Option on Windows* OS
`-opt-report-help`	Displays the optimizer phases available for report generation.	OFF	`/Qopt-report-help`
`-opt-report-phase=`*`phase`*	Specifies an optimizer phase to use when optimization reports are generated.	OFF	`/Qopt-report-phase:`*`phase`*
`-opt-report-routine=`*`string`*	Tells the compiler to generate reports on the routines containing specified text.	OFF	`/Qopt-report-routine:`*`string`*
`-opt-streaming-stores` *`keyword`* (i32, i64em)	Enables generation of streaming stores for optimization.	`-opt-streaming-stores auto`	`/Qopt-streaming-stores:`*`keyword`* (i32, i64em)
`-[no-]opt-subscript-in-range` (i32, i64em)	Determines whether the compiler assumes no overflows in the intermediate computation of subscript expressions in loops.	`-no-opt-subscript-in-range`	`/Qopt-subscript-in-range[-]` (i32, i64em)
`-p`	Compiles and links for function profiling with gprof(1).	OFF	None
`-P`	Causes the Fortran preprocessor to send output to a file, which is named by	OFF	`/P`

Option	Description	Default	Equivalent Option on Windows* OS
	default; same as the `-preprocess-only` option.		
`-[no]pad`	Enables the changing of the variable and array memory layout.	OFF	`/Qpad[-]`
`-[no]pad-source`	Specifies padding for fixed-form source records.	OFF	`/[no]pad-source` or `/Qpad-source[-]`
`-par-affini-ty=[`*`modifier`*`,...]` *`type`*`[,`*`permute`*`] [,`*`offset`*`]` (Linux* OS only)	Specifies thread affinity.	OFF	`/Qpar-affini-ty:[`*`modifier`*`,...]` *`type`*`[,`*`permute`*`][,`*`offset`*`]`
`-par-num-threads=`*`n`*	Specifies the number of threads to use in a parallel region.	OFF	`/Qpar-num-threads:`*`n`*
`-par-report[`*`n`*`]`	Controls the diagnostic information reported by the auto-parallelizer.	`-par-report1`	`/Qpar-report[`*`n`*`]`
`-[no-]par-run-time-control`	Generates code to perform run-time checks for loops that have symbolic loop bounds.	`-no-par-runtime-control`	`/Qpar-runtime-control[-]`
`-par-sched-ule-`*`keyword`* `[=`*`n`*`]`	Specifies a scheduling algorithm for DO loop iterations.	OFF	`/Qpar-sched-ule-`*`keyword`* `[[:]`*`n`*`]`

Option	Description	Default	Equivalent Option on Windows* OS
`-par-threshold[`*n*`]`	Sets a threshold for the auto-parallelization of loops.	`-par-threshold100`	`/Qpar-thresh-old[[:]`*n*`]`
`-parallel`	Tells the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.	OFF	`/Qparallel`
`-pc`*n* (i32, i64em)	Enables control of floating-point significand precision.	`-pc80`	`/Qpc`*n* (i32, i64em)
`-pg`	Compiles and links for function profiling with gprof(1); same as the `-p` option.	OFF	None
`-pie` (Linux OS only)	Produces a position-independent executable on processors that support it.	OFF	None
`-[no-]prec-div`	Improves precision of floating-point divides.	`-prec-div`	`/Qprec-div[-]`
`-[no-]prec-sqrt` (i32, i64em)	Improves precision of square root implementations.	`-no-prec-sqrt`	`/Qprec-sqrt[-]` (i32, i64em)

Option	Description	Default	Equivalent Option on Windows* OS
`-prefetch`	Enables prefetch insertion optimization. Deprecated; use `-opt-prefetch`.	i64: `-prefetch` i32, i64em: `-no-prefetch`	`/Qprefetch`
`-preprocess-only`	Causes the Fortran preprocessor to send output to a file, which is named by default; same as the `-P` option.	OFF	`/preprocess-only`
`-print-multi-lib`	Prints information about where system libraries should be found.	OFF	None
`-[no-]prof-data-order` (Linux OS only)	Enables or disables data ordering if profiling information is enabled.	`-no-prof-data-order`	`/Qprof-data-order[-]`
`-prof-dir` *dir*	Specifies a directory for profiling information output files.	OFF	`/Qprof-dir` *dir*
`-prof-file` *file*	Specifies a file name for the profiling summary file.	OFF	`/Qprof-file` *file*
`-[no-]prof-func-groups` (i32, i64em; Linux OS only)	Enables or disables function grouping if profiling information is enabled.	`-no-prof-func-groups`	None

Option	Description	Default	Equivalent Option on Windows* OS
`-[no-]prof-func-order` (Linux OS only)	Enables or disables function ordering if profiling information is enabled.	`-no-prof-func-order`	`/Qprof-func-order[-]`
`-prof-gen`[`=`*keyword*]	Produces an instrumented object file that can be used in profile-guided optimization.	`-[no-]prof-gen`	`/Qprof-gen`[`:`*keyword*
`-prof-genx`	Produces an instrumented object file that includes extra source position information. Deprecated; use `-prof-gen=srcpos`.	OFF	`/Qprof-genx`
`-prof-hotness-threshold=`*n* (Linux OS only)	Lets you set the hotness threshold for function grouping and function ordering.	OFF	`/Qprof-hotness-threshold:`*n*
`-[no-]prof-src-dir`	Determines whether directory information of the source file under compilation is considered when looking up profile data records.	`-prof-src-dir`	`/Qprof-src-dir[-]`

Option	Description	Default	Equivalent Option on Windows* OS
-prof-src-root=*dir*	Lets you use relative directory paths when looking up profile data and specifies a directory as the base.	OFF	/Qprof-src-root:*dir*
-prof-src-root-cwd	Lets you use relative directory paths when looking up profile data and specifies the current working directory as the base.	OFF	/Qprof-src-root-cwd
-prof-use[=*arg*]	Enables the use of profiling information during optimization.	-no-prof-use	/Qprof-use[:*arg*]
-Qinstall *dir*	Specifies the root directory where the compiler installation was performed.	OFF	None
-Qloca-tion,*string*,*dir*	Specifies a directory as the location of the specified tool in *string* .	OFF	/Qloca-tion,*string*,*dir*
-Qop-tion,*string*,*options*	Passes *options* to the specified tool in *string*.	OFF	/Qop-tion,*string*,*options*
-r8, -r16	Specifies the default KIND for real and complex variables. -r8 is the same as	OFF	/4R8, /4R16

Option	Description	Default	Equivalent Option on Windows* OS
	`/real-size:64`. `-r16` is the same as `/real-size:128`.		
`-rcd` (i32, i64em)	Enables fast float-to-integer conversions.	OFF	`/Qrcd` (i32, i64em)
`-rct` (i32, i64em)	Sets the internal FPU rounding control to Truncate.	OFF	`/Qrct` (i32, i64em)
`-real-size` *size*	Specifies the default KIND for real variables.	`-real-size 32`	`/real-size:`*size*
`-recursive`	Tells the compiler that all routines should be compiled for possible recursive execution.	`-norecursive`	`/recursive`
`-reentrancy` *keyword*	Tells the compiler to generate reentrant code to support a multithreaded application.	`-noreentrancy`	`/reentrancy:`*keyword*
`-S`	Causes the compiler to compile to an assembly file (.s) only and not link; same as options `/Fa` and `/asmfile`.	OFF	`/S`

Option	Description	Default	Equivalent Option on Windows* OS
-safe-cray-ptr	Tells the compiler that Cray* pointers do not alias other variables.	OFF	/Qsafe-cray-ptr
-save	Causes variables to be placed in static memory.	-auto-scalar	/Qsave
-[no-]save-temps	Tells the compiler to save intermediate files created during compilation.	-no-save-temps	/Qsave-temps[-]
-[no-]scalar-rep (i32 only)	Enables scalar replacement performed during loop transformation (requires -O3).	-no-scalar-rep	/Qscalar-rep[-] (i32 only)
-shared (Linux OS only)	Tells the compiler to produce a dynamic shared object instead of an executable.	OFF	None
-shared-intel	Links Intel-provided libraries dynamically.	OFF	None
-shared-libgcc (Linux OS only)	Links the GNU libgcc library dynamically.	-shared-libgcc	None
-[no-]sox (Linux OS only)	Tells the compiler to save the compilation options and version number in the Linux OS executable.	-no-sox	/Qsox[-]

Option	Description	Default	Equivalent Option on Windows* OS
`-stand` *keyword*	Causes the compiler to issue compile-time messages for nonstandard language elements.	`-nostand`	`/stand:`*keyword*
`-static` (Linux OS only)	Prevents linking with shared libraries.	`-static`	`/static`
`-staticlib` (i32, i64em; Mac OS X only)	Invokes the libtool command to generate static libraries.	OFF	None
`-static-intel`	Links Intel-provided libraries statically.	OFF	None
`-static-libgcc` (Linux OS only)	Links the GNU libgcc library statically.	OFF	None
`-std90`	Causes the compiler to issue messages for language elements that are not standard in Fortran 90; same as `-stand f90`.	OFF	`/stand:f90`
`-std95`	Causes the compiler to issue messages for language elements that are not standard in Fortran 95; same as `-stand f95`.	OFF	`/stand:f95`
`-std03`	Causes the compiler to issue messages for language elements	OFF	`/stand:f03`

Option	Description	Default	Equivalent Option on Windows* OS
	that are not standard in Fortran 2003; same as `-std` or `-stand f03`.		
`-syntax-only`	Specifies that the source file should be checked only for correct syntax.	OFF	`/syntax-only`
`-T` *file* (Linux OS only)	Tells the linker to read link commands from the specified *file*.	OFF	None
`-tcheck` (Linux OS only)	Enables analysis of threaded applications.	OFF	`/Qtcheck`
`-tcollect` [*lib*] (Linux OS only)	Inserts instrumentation probes calling the Intel(R) Trace Collector API.	OFF	`/Qtcollect`[=*lib*]
`-tcollect-filter` [*file*] (Linux OS only)	Lets you enable or disable the instrumentation of specified functions.	OFF	`/Qtcollect-filter`[=*file*]
`-Tf` *file*	Tells the compiler to compile the file as a Fortran source file.	OFF	`/Tf` *file*

Option	Description	Default	Equivalent Option on Windows* OS
`-[no]threads`	Tells the linker to search for unresolved references in a multithreaded run-time library.	i32, i64: `-nothreads` i64em: `-threads`	`/[no]threads`
`-tprofile` (Linux OS only)	Generates instrumentation to analyze multi-threading performance.	OFF	`/Qtprofile`
`-[no]traceback`	Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.	`-notraceback`	`/[no]traceback`
`-tune` *keyword* (i32, i64em)	Determines the version of the architecture for which the compiler generates instructions.	`-tune pn4`	`/tune:`*keyword* (i32, i64em)
`-u`	Enables error messages about any undeclared symbols; same as the `-warn declarations` option.	OFF	None Note: the Windows option `/u` is not the same

Option	Description	Default	Equivalent Option on Windows* OS
-U*name*	Undefines any definition currently in effect for the specified symbol.	OFF	/U*name*
-unroll[*n*]	Tells the compiler the maximum number of times to unroll loops. -unroll is the same as option -funroll-loops.	-unroll	/unroll[:*n*]
-[no-]unroll-aggressive (i32, i64em)	Determines whether the compiler uses more aggressive unrolling for certain loops.	-no-unroll-aggressive	/Qunroll-aggressive[-] (i32, i64em)
-uppercase	Causes the compiler to ignore case differences in identifiers and to convert external names to uppercase; same as the -names uppercase option.	OFF	/Quppercase
-us	Tells the compiler to append an underscore character to external user-defined names; same as the -assume underscore option.	-us	/us

Option	Description	Default	Equivalent Option on Windows* OS
`-[no-]use-asm`	Tells the compiler to produce objects through the assembler.	`-no-use-asm`	`/Quse-asm[-]` (i32 only)
`-v` [*file*]	Tells the driver that tool commands should be shown and executed.	OFF	None
`-V`	Displays the compiler version information; same as the `-logo` option.	OFF	None
`-[no-]vec` (i32, i64em)	Enables or disables vectorization and transformations enabled for vectorization.	`-no-vec`	`/Qvec[-]` (i32, i64em)
`-[no-]vec-guard-write` (i32, i64em)	Tells the compiler to perform a conditional check in a vectorized loop.	`-no-vec-guard-write`	`/Qvec-guard-write[-]` (i32, i64em)
`-vec-report`[*n*] (i32, i64em)	Controls the diagnostic information reported by the vectorizer.	`-vec-report1`	`/Qvec-report`[*n*] (i32, i64em)
`-vec-threshold`[*n*] (i32, i64em)	Sets a threshold for the vectorization of loops.	`-vec-threshold100`	`/Qvec-thresh-old`[[:]*n*] (i32, i64em)

Option	Description	Default	Equivalent Option on Windows* OS
-[no]vms	Causes the run-time system to behave like HP* Fortran on OpenVMS* Alpha systems and VAX* systems (VAX FORTRAN*).	-novms	/[no]vms
-w	Disables all warning messages; same as specifying option -warn noneor -warn nogeneral.	OFF	/w
-W*n*	Disables (*n*=0) or enables (*n*=1) all warning messages.	-W1	/W*n*
-Wa,o1[,o2,...]	Passes options (o1,o2, and so forth) to the assembler for processing.	OFF	None
-warn [*keyword*]	Specifies diagnostic messages to be issued by the compiler.	keywords: alignments nodeclarations noerrors general noignore_loc nointerfaces nostderrors notruncated_source nouncalled nounused usage	/warn[:*keyword*]

Option	Description	Default	Equivalent Option on Windows* OS
-[no]watch [*keyword*]	Tells the compiler to display certain information to the console output window.	-nowatch	/[no]watch[:*keyword*]
-WB	Turns a compile-time bounds check error into a warning.	OFF	/WB
-what	Tells the compiler to display its detailed version string.	OFF	/what
-Winline	Enables diagnostics about what is inlined and what is not inlined.	OFF	None
-Wl,option1 [,option2,...]	Passes options (o1, o2, and so forth) to the linker for processing.	OFF	None
-Wp,option1 [,option2,...]	Passes options (o1, o2, and so forth) to the preprocessor.	OFF	None
-x*p* (i32, i64em)	Tells the compiler to generate optimized code specialized for the Intel processor that executes your program.	varies; see the option description	/Qx*p* (i32, i64em)

Option	Description	Default	Equivalent Option on Windows* OS
-X	Removes standard directories from the include file search path.	OFF	/X
-Xlinker *option*	Passes a linker option directly to the linker	OFF	None
-y	Specifies that the source file should be checked only for correct syntax; same as the -syntax-only option.	OFF	/Zs
-[no]zero	Initializes to zero all local scalar variables of intrinsic type INTEGER, REAL, COMPLEX, or LOGICAL that are saved but not yet initialized.	-nozero	/Qzero[-]
-Zp[*n*]	Aligns fields of records and components of derived types on the smaller of the size boundary specified or the boundary that will naturally align them.	-Zp16	/Zp[*n*]

See Also

- Quick Reference Guides and Cross References

- `-map-opts`, `/Qmap-opts`

Related Options

22

This topic lists related options that can be used under certain conditions.

Linking Tools and Options

This topic describes how to use the Intel® linking tools, `xild` (Linux* OS and Mac OS* X) or `xilink` (Windows* OS).

The Intel linking tools behave differently on different platforms. The following sections summarizes the primary differences between the linking behaviors.

Linux OS and Mac OS X Linking Behavior Summary

The linking tool invokes the compiler to perform IPO if objects containing IR (intermediate representation) are found. (These are mock objects.) It invokes GNU `ld` to link the application.

The command-line syntax for `xild` is the same as that of the GNU linker:

```
xild [<options>] <normal command-line>
```

where:

- `[<options>]`: (optional) one or more options supported only by `xild`.
- `<normal command-line>`: linker command line containing a set of valid arguments for `ld`.

To create `app` using IPO, use the option `-o`*`file`* as shown in the following example:

```
xild -qipo_fas -oapp a.o b.o c.o
```

The linking tool calls the compiler to perform IPO for objects containing IR and creates a new list of object(s) to be linked. The linker then calls `ld` to link the object files that are specified in the new list and produce the application with the name specified by the `-o` option. The linker supports the `-ipo`, `-ipo`*`N`*, and `-ipo-separate` options.

Windows OS Linking Behavior Summary

The linking tool invokes the Intel compiler to perform multi-file IPO if objects containing IR (intermediate representation) is found. These are mock objects. It invokes Microsoft* `link.exe` to link the application.

Windows OS Linking Behavior Summary

The command-line syntax for the Intel® linker is the same as that of the Microsoft linker:
`xilink [<options>] <normal command-line>`
where:
• `[<options>]`: (optional) one or more options supported only by `xilink`.
• `<normal command-line>`: linker command line containing a set of valid arguments for the Microsoft linker.
To place the multifile IPO executable in `ipo_file.exe`, use the linker option `/out:`*`file`*; for example:
`xilink -qipo_fas /out:ipo_file.exe a.obj b.obj c.obj`
The linker calls the compiler to perform IPO for objects containing `IR` and creates a new list of object(s) to be linked. The linker calls Microsoft `link.exe` to link the object files that are specified in the new list and produce the application with the name specified by the `/out:`*`file`* linker option.

Using the Linking Tools

You must use the Intel linking tools to link your application if the following conditions apply:

- Your source files were compiled with multifile IPO enabled. Multi-file IPO is enabled by specifying the `-ipo` (Linux OS and Mac OS X) or `/Qipo` (Windows OS) command-line option.
- You normally would invoke either the GNU linker (`ld`) or the Microsoft linker (`link.exe`) to link your application.

The following table lists the available, case-insensitive options supported by the Intel linking tools and briefly describes the behavior of each option:

Linking Tools Option	Description
`-qhelp`	Lists the available linking tool options. Same as passing no option.
`-qnoipo`	Disables multi-file IPO compilation.

Linking Tools Option	Description
-qipo_fa*[{file|dir/}]*	Produces assembly listing for the multi-file IPO compilation. You may specify an optional name for the listing file, or a directory (with the backslash) in which to place the file. The default listing name is depends on the platform: • Linux OS and Mac OS X: ipo_out.s • Windows OS: ipo_out.asm If the Intel linking tool invocation results in multi-object compilation, either because the application is big or because the user explicitly instructed the compiler to generate multiple objects, the first .s (Linux OS and Mac OS X) or .asm (Windows OS) file takes its name from the -qipo_fa option. The compiler derives the names of subsequent .s (Linux OS and Mac OS X) or .asm (Windows OS) files by appending an incrementing number to the name, for example, foo.asm and foo1.asm for ipo_fafoo.asm. The same is true for the -qipo_fo option (listed below).
-qipo_fo*[{file|dir/}]*	Produces object file for the multi-file IPO compilation. You may specify an optional name for the object file, or a directory (with the backslash) in which to place the file. The default object file name is depends on the platform: • Linux OS and Mac OS X: ipo_out.o • Windows OS: ipo_out.obj
-qipo_fas	Add source lines to assembly listing.

Linking Tools Option	Description
-qipo_fac	Adds code bytes to the assembly listing.
-qipo_facs	Add code bytes and source lines to assembly listing.
-quseenv	Disables override of existing PATH, LIB, and INCLUDE variables.
-lib	Invokes librarian instead of linker.
-libtool	Mac OS X: Invokes libtool to create a library instead of ld.
-qv	Displays version information.

See Also

- Related Options
- Using IPO

Fortran Preprocessor Options

The Fortran preprocessor (fpp) may be invoked automatically or by specifying option fpp.

The following options are available if fpp is in effect.

fpp Option	Description
-B	Specifies that C++-style comments should not be recognized.
-C	Specifies that C-style comments should not be recognized. This is the same as specifying -c_com=no.

fpp Option	Description
`-c_com={yes|no}`	Determines whether C-style comments are recognized. If you specify `-c_com=no` or `-C`, C-style comments are not recognized. By default, C-style comments are recognized; that is, `-c_com=yes`.
`-Dname`	Defines the preprocessor variable name as 1 (one). This is the same as if a `-Dname=1` option appeared on the fpp command line, or as if a `#define name 1` line appeared in the source file processed by fpp.
`-Dname=def`	Defines name as if by a #define directive. This is the same as if a `#define name def` line appeared in the source file processed by fpp. The `-D` option has lower precedence than the `-U` option. That is, if the same name is used in both a `-U` option and a `-D` option, the name will be undefined regardless of the order of the options.
`-e`	Tells the compiler to accept extended source lines. For fixed format, lines can contain up to 132 characters. For free format, lines can contain up to 32768 characters.
`-e80`	Tells the compiler to accept extended source lines. For fixed format, lines can contain up to 80 characters.
`-fixed`	Tells the compiler to assume fixed format in the source file.

fpp Option	Description
-free	Tells the compiler to assume free format in the source file.

-f_com={yes|no}	Determines whether Fortran-style end-of-line comments are recognized or ignored by fpp. If you specify -f_com=no, Fortran style end-of-line comments are processed as part of the preprocessor directive. By default, Fortran style end-of-line comments are recognized by fpp on preprocessor lines and are ignored by fpp; that is, -f_com=yes. For example:
	`#define max 100 ! max number`
	`do i = 1, max + 1`
	If you specify -f_com=yes, fpp will output
	`do i = 1, 100 + 1`
	If you specify -f_com=no, fpp will output
	`do i = 1, 100 ! max number + 1`

-help	Displays information about fpp options.
-I<dir>	Inserts directory <dir> into the search path for #include files with names not beginning with "/". The <dir> is inserted ahead of the standard list of "include" directories so that #include files with names enclosed in double-quotes (") are searched for first in the directory of the file with the #include line, then in directories named with -I options, and lastly, in directories from the standard list. For #include files with names enclosed in angle-brackets (<>), the directory of the file with the #include line is not searched.
-m	Expands macros everywhere. This is the same as -macro=yes.

fpp Option	Description
macro={yes|no_com|no}	Determines the bahavior of macro expansion. If you specify -macro=no_com, macro expansion is turned off in comments. If you specify -macro=no, no macro expansion occurs anywhere. By default, macros are expanded everywhere; that is, -macro=yes.
-noB	Specifies that C++-style comments should be recognized.
-noC	Specifies that C-style comments should be recognized. This is the same as -c_com=yes.
-noJ	Specifies that F90-style comments should be recognized in a #define line. This is the same as -f_com=no.
-no-fort-cont	Specifies that IDL style format should be recognized. This option is only for the IDE. Note that macro arguments in IDL may have a C-like continuation character "\" which is different from the Fortran continuation character "&". Fpp should recognize the C-like continuation character and process some other non-Fortran tokens so that the IDL processor can recognize them.
-P	Tells the compiler that line numbering directives should not be added to the output file. This line-numbering directive appears as #line-number file-name
-Uname	Removes any initial definition of name, where name is an fpp variable that is predefined on a particular preprocessor. Here is a partial list of symbols that may be predefined, depending upon the architecture of the system:

fpp Option	Description
	Operating System: __APPLE__, __unix, and __linux Hardware: __i386, __ia64, __x86_64
-undef	Removes initial definitions for all predefined symbols.
-V	Displays the fpp version number.
-w[0]	Prevents warnings from being output. By default, warnings are output to stderr.
-Xu	Converts uppercase letters to lowercase, except within character-string constants. The default is to leave the case as is.
-Xw	Tells the compiler that in fixed-format source files, the blank or space symbol " " is insignificant. By default, the space symbol is the delimiter of tokens for this format.
-Y<dir>	Adds directory <dir> to the end of the system include paths.

For details on how to specify these options on the compiler command line, see fpp, Qfpp.

Floating-Point Environment

The floating-point (FP) environment is a collection of registers that control the behavior of FP machine instructions and indicate the current FP status. The floating-point environment may include rounding mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.

Part III

Optimizing Applications

Topics:

- Intel(R) Fortran Optimizing Applications
- Evaluating Performance
- Using Compiler Optimizations
- Using Parallelism: OpenMP* Support
- Using Parallelism: Automatic Parallelization
- Using Parallelism: Automatic Vectorization
- Using Parallelism: Multi-Threaded Applications
- Using Interprocedural Optimization (IPO)
- Using Profile-Guided Optimization (PGO)
- Using High-Level Optimization (HLO)
- Optimization Support Features
- Programming Guidelines

Intel(R) Fortran Optimizing Applications

23

Overview: Optimizing Applications

Optimizing Applications explains how to use the Intel® Fortran Compiler to help improve application performance.

How you use the information presented in this document depends on what you are trying to accomplish. You can start with the following topics:

- Optimizing with the Intel Compiler
- Optimizing for Performance
- Overview of Parallelism Methods
- Quick Reference Lists
- Other Resources
- Performance Analysis

Where applicable this document explains how compiler options and optimization methods differ on IA-32, Intel® 64, and IA-64 architectures on Linux* operating systems (OS), Intel®-based systems running Mac OS* X, and Windows* operating systems.

While the compiler supports Integrated Development Environments (IDE) on several different operating systems, the concepts and examples included here illustrate using the compiler from the command line.

In most cases, the compiler features and options supported for IA-32 or Intel® 64 architectures on Linux OS are also supported on Intel-based systems running Mac OS X. For more detailed information about support for specific operating systems, refer to the appropriate option in Compiler Options.

Optimizing with the Intel® Compiler

The Intel compiler supports a variety of options and features that allow optimization opportunities; however, in most cases you will benefit by applying optimization strategies in the order listed below.

Use Automatic Optimizations

Use the automatic optimization options, like `-O1`, `-O2`, `-O3`, or `-fast` (Linux* and Mac OS* X) and `/O1`, `/O2`, `/O3`, or `/fast` (Windows*) to determine what works best for your application. Use these options and measure the resulting performance after each compilation.

NOTE. The optimizer that integrates parallelization (IA-32, Intel® 64, and IA-64 architectures) and vectorization (IA-32 and Intel® 64 architectures) has been redesigned to provide performance improvements when specifying the `-O2` or `-O3` (Linux and Mac OS X) and `/O2` or `/O3` (Windows) options.

You might find specific options to work better on a particular architecture:

- IA-32 and Intel® 64 architectures: start with `-O2` (Linux and Mac OS X) or `/O2` (Windows).
- IA-64 architecture: start with `-O3` (Linux and Mac OS X) or `/O3` (Windows).

If you plan to run your application on specific architectures, experiment with combining the automatic optimizations with compiler options that specifically target processors.

You can combine the `-x` and `-ax` (Linux* and Mac OS* X) or `/Qx` and `/Qax` (Windows*) options to generate both code that is optimized for specific Intel processors and generic code that will run on most processors based on IA-32 and Intel® 64 architectures. See the following topics for more information about targeting processors:

- Targeting IA-32 and Intel® 64 Architecture Processors Automatically
- Targeting Multiple IA-32 and Intel 64 Architecture Processors Automatically for Run-time Performance
- Targeting Itanium® Processors Automatically

Attempt to combine the automatic optimizations with the processor-specific options before applying other optimizations techniques.

Use IPO and PGO

Experiment with Interprocedural Optimization (IPO) and Profile-guided Optimization (PGO). Measure performance after applying the optimizations to determine whether the application performance improved.

Use a top-down, iterative method for identifying and resolving performance-hindering code using performance monitoring tools, like the compiler reports.

Use Parallelism

If you are planning to run the application on multi-core or multi-processor systems, start the parallelism process by using the parallelism options or OpenMP* options.

Compiling for Older Processors

Use automatic optimization options and other processor-independent compiler options to generate optimized code that do not take advantage of advances in processor design or extension support. Use the `-x` (Linux* and Mac OS* X) or `/Qx` (Windows*) option to generate processor dispatch for older processors.

Optimizing for Performance

The following table lists possible starting points for your optimization efforts.

If you are trying to...	Start with these topics or sections.
use performance analysis to begin the optimization process	• Optimizing with Intel® Compilers • Using a Performance Enhancement Methodology • Intel® Performance Analysis Tools and Libraries • Performance Enhancement Strategies
optimize for speed or a specific architecture	• Enabling Automatic Optimizations • Targeting IA-32 and Intel® 64 Architecture Processors Automatically • Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance • Targeting IA-64 Architecture Processors Automatically
create parallel programs or parallelize existing programs	• Using Parallelism • Automatic Vectorization Overview • OpenMP* Support Overview

If you are trying to...	Start with these topics or sections.
	• Auto-parallelization Overview
use Interprocedural Optimization	• Using IPO • IPO for Large Programs • Interprocedural Optimization (IPO) Quick Reference
create application profiles to help optimization	• Profile an Application • Profile-Guided Optimization (PGO) Quick Reference • profmerge and proforder Tools • PGO API and environment variables
generate reports on compiler optimizations	• Generating Reports • Compiler Reports Quick Reference
optimize loops, arrays, and data layout	• High-level Optimization (HLO) Overview
use programming strategies to improve performance	• Setting Data Type and Alignment • Using Arrays Efficiently • Improving I/O Performance • Improving Run-time Efficiency • Using Intrinsics for IA-64 architecture based Systems • Coding Guidelines for Intel Architectures

Overview of Parallelism Method

The three major features of parallel programming supported by the Intel® compiler include:

- OpenMP*
- Auto-parallelization

- Auto-vectorization

Each of these features contributes to application performance depending on the number of processors, target architecture (IA-32, Intel® 64, and IA-64 architectures), and the nature of the application. These features of parallel programming can be combined to contribute to application performance.

Parallelism defined with the OpenMP* API is based on thread-level and task-level parallelism. Parallelism defined with auto-parallelization techniques is based on thread-level parallelism (TLP). Parallelism defined with auto-vectorization techniques is based on instruction-level parallelism (ILP).

Parallel programming can be *explicit*, that is, defined by a programmer using the OpenMP* API and associate options. Parallel programming can also be *implicit*, that is, detected automatically by the compiler. Implicit parallelism implements auto-parallelization of outer-most loops and auto-vectorization of innermost loops (or both).

To enhance the compilation of the code with auto-vectorization, users can also add vectorizer directives to their program.

NOTE. Software pipelining (SWP), a technique closely related to auto-vectorization, is available on systems based on IA-64 architecture.

The following table summarizes the different ways in which parallelism can be exploited with the Intel® Compiler.

Intel provides performance libraries that contain highly optimized, extensively threaded routines, including the Intel® Math Kernel Library (Intel® MKL).

In addition to these major features supported by the Intel compiler, certain operating systems support application program interface (API) function calls that provide explicit threading controls. For example, Windows* operating systems support API calls such as `CreateThread`, and multiple operating systems support POSIX* threading APIs.

Parallelism Method	Supported On
Implicit (parallelism generated by the compiler and by user-supplied hints)	
Auto-parallelization (Thread-Level Parallelism)	• IA-32 architecture, Intel® 64 architecture, IA-64 architecture based multi-processor systems, and multi-core processors • Hyper-Threading Technology-enabled systems

Parallelism Method	Supported On
Auto-vectorization (Instruction-Level Parallelism)	• Pentium®, Pentium with MMX™ Technology, Pentium II, Pentium III, Pentium 4 processors, Intel® Core™ processor, and Intel® Core™ 2 processor.
Explicit (parallelism programmed by the user)	
OpenMP* (Thread-Level and Task-Level Parallelism)	• IA-32 architecture, Intel® 64 architecture, IA-64 architecture-based multiprocessor systems, and multi-core processors • Hyper-Threading Technology-enabled systems

Threading Resources

For general information about threading an existing serial application or design considerations for creating new threaded applications, see Other Resources and the web site http://go-parallel.com.

Quick Reference Lists

There are several quick reference guides in this document. Use these quick reference guides to quickly familiarize yourself with the compiler options or features available for specific optimizations.

- Enabling Automatic Optimizations
- Interprocedural Optimization (IPO) Quick Reference
- Profile-guided Optimization (PGO) Quick Reference
- Auto-Vectorization Options Quick Reference
- OpenMP* Options Quick Reference
- Auto-Parallelization Options Quick Reference
- Compiler Reports Quick Reference
- Data Alignment Options
- PGO Environment Variables
- OpenMP* Environment Variables

Other Resources

Understanding the capabilities of the specific processors and the underlying architecture on which your application will run is key to optimization. Intel distributes many hardware and software development resources that can help better understand how to optimize application source code for specific architectures.

Processor Information

You can find detailed information about processor numbers, capabilities, and technical specifications, along with documentation, from the following web sites:

- *Intel® Processor Spec Finder* (http://processorfinder.intel.com/)
- *Intel® Processor Numbers* (http://www.intel.com/products/processor_number/)
- *Intel® Processor Identification Utility* (http://www.intel.com/support/processors/tools/piu/)

Architecture Information

The architecture manuals provide specific details about the basic architecture, supported instruction sets, programming guidelines for specific operating systems, and performance monitoring.

The optimization manuals provide insight for developing high-performance applications for Intel® architectures.

- IA-32 and Intel® 64 architectures: http://www.intel.com/products/processor/manuals/index.htm
- IA-64 Architecture: http://www.intel.com/design/itanium/documentation.htm

Optimization Strategy Resources

For more information on advanced or specialized optimization strategies, refer to the Resource Centers for Software Developers, which can be accessed from www.intel.com. Refer to the articles, community forums, and links to additional resources in the listed areas of the following Developer Centers:

Tools and Technologies:

- Threading/Multi-core

- Intel® Software Products

Intel® Processors:

- Intel® 64 and IA-32 Architectures Software Developer's Manuals
- Itanium® Processor Family
- Pentium® 4 Processor
- Intel® Xeon® Processor

Environments:

- High Performance Computing

Evaluating Performance 24

Performance Analysis

The high-level information presented in this section discusses methods, tools, and compiler options used for analyzing runtime performance-related problems and increasing application performance.

The topics in this section discuss performance issues and methods from a general point of view, focusing primarily on general performance enhancements. The information in this section is separated in the following topics:

- Using a Performance Enhancement Methodology
- Performance Enhancement Strategies
- Using Intel Performance Analysis Tools

In most cases, other sections and topics in this document contain detailed information about the options, concepts, and strategies mentioned in this section.

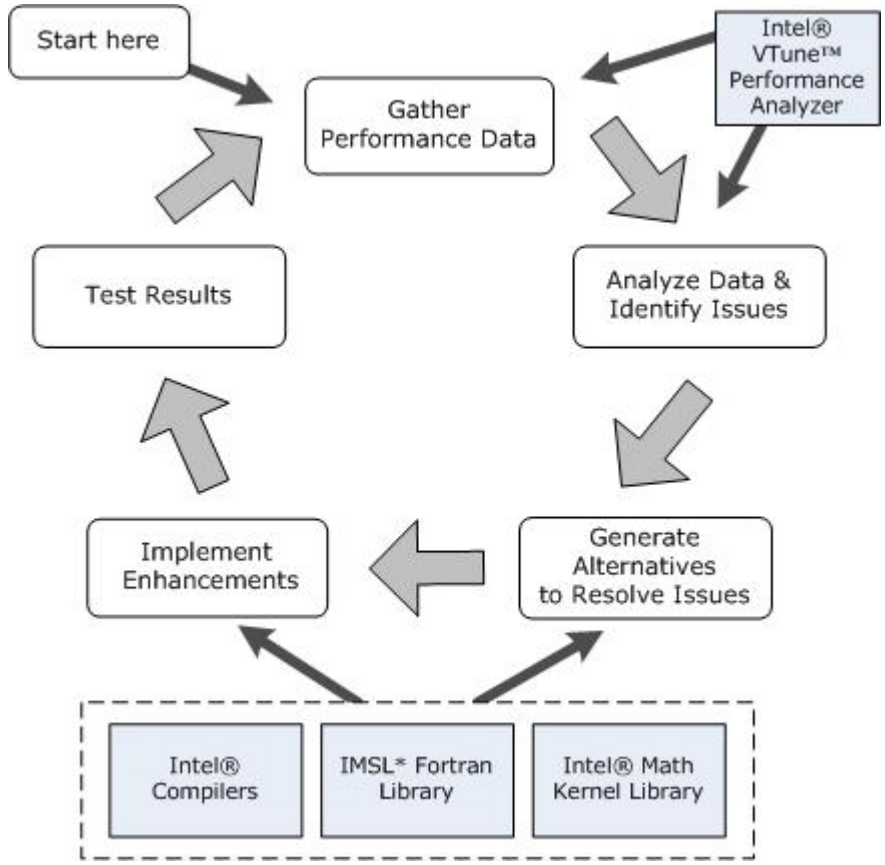
This section also contains information about using the compiler-supporting reports and diagnostics.

Using a Performance Enhancement Methodology

The recommended performance enhancement method for optimizing applications consists of several phases. When attempting to identify performance issues, move through the following general phases in the order presented:

- Gather performance data
- Analyze the data
- Generate alternatives
- Implement enhancements
- Test the results

The following figure shows the methodology phases and their relationships, along with some recommended tools to use in each appropriate phase.



The methodology can be summarized by the following statements:

- Make small changes and measure often.
- If you approach a point of diminishing return and can find no other performance issues, stop optimizing.

Gather performance data

Use tools to measure where performance bottlenecks occur; do not waste time guessing. Using the right tools for analysis provides an objective data set and baseline criteria to measure implementation changes and improvements introduced in the other stages.

See Using Intel Performance Analysis Tool and Libraries for more information about some tools you can use to gather performance data.

Analyze the data

Determine if the data meet your expectations about the application performance. If not, choose one performance problem at a time for special interest. Limiting the scope of the corrections is critical in effective optimization.

In most cases, you will get the best results by resolving hotspots first. Since hotspots are often responsible for excessive activity or delay, concentrating on these areas tends to resolve or uncover other performance problems that would otherwise be undetectable.

Generate alternatives

As in the analysis phase, limit the focus of the work. Concentrate on generating alternatives for the one problem area you are addressing. Identify and use tools and strategies to help resolve the issues. For example, you can use compiler optimizations, use Intel® Performance Library routines, or use some other optimization (like improved memory access patterns, reducing or eliminating division or other floating-point operations, rewriting the code to include intrinsics or assembly code, or other strategies).

See Performance Enhancement Strategies for suggestions.

While optimizing for the compiler and source levels, consider using the following strategies in the order presented:

1. Use available supported compiler options. This is the most portable, least intrusive optimization strategy.
2. Use compiler directives embedded in the source. This strategy is not overly intrusive since the method involves including a single line in code, which can be ignored (optionally) by the compiler.
3. Attempt manual optimizations.

Implement enhancements

As with the previous phases, limit the focus of the implementation. Make small, incremental changes. Trying to address too many issues at once can defeat the purpose and reduce your ability to test the effectiveness of your enhancements.

The easiest enhancements will probably involve enabling common compiler optimizations for easy gains. For applications that can benefit from the libraries, consider implementing Intel® Performance Library routines that may require some interface coding.

Test the results

If you have limited the scope of the analysis and implementation, you should see measurable differences in performance in this phase. Have a target performance level in mind so you know when you have reached an acceptable gain in performance.

Use a consistent, reliable test that reports a quantifiable item, like seconds elapsed, frames per second, and so forth, to determine if the implementation changes have actually helped performance.

If you think you can make significant improvement gains or you still have other performance issues to address, repeat the phases beginning with the first one: gather performance data.

Intel® Performance Analysis Tools and Libraries

Intel Corporation offers a variety of performance analysis tools and libraries that can help you optimize your application performance.

Performance Analysis Tools

These performance tools can help you analyze your application, find problem areas, and develop efficient programs. In some cases, these tools are critical to the optimization process.

Tool	Operating System	Description
Intel® Threading Analysis Tools	Linux, Windows	Intel® Threading Analysis Tools consist of the Intel® Thread Checker and the Intel® Thread Profiler. The Intel® Thread Checker can help identify shared and private variable conflicts, and can isolate threading bugs to the source code line where the bug occurs.

Tool	Operating System	Description
		The Intel® Thread Profiler can show the critical path of an application as it moves from thread to thread, and identify synchronization issues and excessive blocking time that cause delays for Win32*, POSIX* threaded, and OpenMP* code.

Performance Libraries

These performance libraries can decrease development time and help to increase application performance.

Library	Operating System	Description
Intel® Math Kernel Library	Linux, Mac OS X, Windows	Intel® Math Kernel Library offers highly optimized, thread-safe math routines for science, engineering, and financial applications that require maximum performance.

Performance Enhancement Strategies

Improving performance starts with identifying the characteristics of the application you are attempting to optimize. The following table lists some common application characteristics, indicates the overall potential performance impact you can expect, and provides suggested solutions to try. These strategies have been found to be helpful in many cases; experimentation is key with these strategies.

In the context of this discussion, view the potential impact categories as an indication of the possible performance increases that might be achieved when using the suggested strategy. It is possible that application or code design issues will prohibit achieving the indicated increases; however, the listed impacts are generally true. The impact categories are defined in terms of the following performance increases, when compared to the initially tested performance:

- Significant: more than 50%
- High: up to 50%
- Medium: up to 25%
- Low: up to 10%

The following table is ordered by application characteristics and then by strategy with the most significant potential impact.

Application Characteristics	Impact	Suggested Strategies
Technnical Applications		
Technical applications with loopy code	High	Technical applications are those programs that have some subset of functions that consume a majority of total CPU cycles in loop nests. Target loop nests using `-O3` (Linux* and Mac OS* X) or `/O3` (Windows*) to enable more aggressive loop transformations and prefetching. Use High-Level Optimization (HLO) reporting to determine which HLO optimizations the compiler elected to apply. See High-Level Optimization Report.
(same as above) IA-64 architecture only	High	For `-O2` and `-O3` (Linux) or `/O2` and `/O3` (Windows), use the swp report to determine if Software Pipelining occurred on key loops, and if not, why not. You might be able to change the code to allow software pipelining under the following conditions:

Application Characteristics	Impact	Suggested Strategies
		• If recurrences are listed in the report that you suspect do not exist, eliminate aliasing problems , or use `IVDEP` directive on the loop. • If the loop is too large or runs out of registers, you might be able to distribute the loop into smaller segments; distribute the loop manually or by using the `distribute` directive. • If the compiler determines the Global Acyclic Scheduler can produce better results but you think the loop should still be pipelined, use the `SWP` directive on the loop.
(same as above) IA-32 and Intel® 64 architectures only	High	See Vectorization Overview and the remaining topics in the Auto-Vectorization section for applicable options. See Vectorization Report for specific details about when you can change code.
(same as above)	Medium	Use PGO profile to guide other optimizations. See Profile-guided Optimizations Overview.
Applications with many denormalized floating-point value operations	Significant	Experiment with `-fp-model fast=2` (Linux and Mac OS X) or `/fp:fast=2` or `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows). The

Application Characteristics	Impact	Suggested Strategies
		resulting performance increases can adversely affect floating-point calculation precision and reproducibility. See Floating-point Operations for more information about using the floating point options supported in the compiler.
Sparse matrix applications	Medium	See the suggested strategy for memory pointer disambiguation (below). Use `prefetch` directive or prefetch intrinsics. Experiment with different prefetching schemes on indirect arrays. See HLO Overview or Data Prefetching starting places for using prefetching.
Server application with branch-centric code and a fairly flat profile	Medium	Flat profile applications are those applications where no single module seems to consume CPU cycles inordinately. Use PGO to communicate typical hot paths and functions to the compiler, so the Intel® compiler can arrange code in the optimal manner. Use PGO on as much of the application as is feasible. See Profile-guided Optimizations Overview.

Application Characteristics	Impact	Suggested Strategies
Database engines	Medium	Use `-O1` (Linux and Mac OS X) or `/O1` (Windows) and PGO to optimize the application code.
Other Application Types		
Applications with many small functions that are called from multiple locations	Low	Use `-ip` (Linux and Mac OS X) or `/Qip` (Windows) to enable inter-procedural inlining within a single source module. Streamlines code execution for simple functions by duplicating the code within the code block that originally called the function. This will increase application size. As a general rule, do not inline large, complicated functions. See Interprocedural Optimizations Overview.
(same as above)	Low	Use `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) to enable inter-procedural inlining both within and between multiple source modules. You might experience an additional increase over using `-ip` (Linux and Mac OS X) or `/Qip` (Windows). Using this option will increase link time due to the extended program flow analysis that occurs.

Application Characteristics	Impact	Suggested Strategies
		Use Interprocedural Optimization (IPO) to attempt to perform whole program analysis, which can help memory pointer disambiguation.

Apart from application-specific suggestions listed above, there are many application-, OS/Library-, and hardware-specific recommendations that can improve performance as suggested in the following tables:

Application-specific Recommendations

Application Area	Impact	Suggested Strategies
Cache Blocking	High	Use `-O3` (Linux and Mac OS X) or `/O3` (Windows) to enable automatic cache blocking; use the HLO report to determine if the compiler enabled cache blocking automatically. If not consider manual cache blocking. See Cache Blocking.
Compiler directives for better alias analysis	Medium	Ignore vector dependencies. Use `IVDEP` and other directives to increase application speed. See Vectorization Support.
Math functions	Low	Use float intrinsics for single precision data type, for example, `sqrtf()` not `sqrt()`. Call Math Kernel Library (MKL) instead of user code. Call F90 intrinsics instead of user code (to enable optimizations).

Library/OS Recommendations

Area	Impact	Description
Symbol preemption	Low	Linux has a less performance-friendly symbol preemption model than Windows. Linux uses full preemption, and Windows uses no preemption. Use `-fminshared -fvisibility=protected`. See Symbol Visibility Attribute Options.
Memory allocation	Low	Using third-party memory management libraries can help improve performance for applications that require extensive memory allocation.

Hardware/System Recommendations

Component	Impact	Description
Disk	Medium	Consider using more advanced hard drive storage strategies. For example, consider using SCSI instead of IDE. Consider using the appropriate RAID level. Consider increasing the number hard drives in your system.
Memory	Low	You can experience performance gains by distributing memory in a system. For example, if you have four open memory slots and only two slots are

Component	Impact	Description
		populated, populating the other two slots with memory will increase performance.
Processor		For many applications, performance scales is directly affected by processor speed, the number of processors, processor core type, and cache size.

Using Compiler Reports

Compiler Reports Overview

The Intel® compiler provides several reports that can help identify performance issues.

Some of these compiler reports are architecture-specific, most are more general. Start with the general reports that are common to all platforms (operating system/architecture), then use the reports unique to an architecture.

- Generating reports
- Interprocedural Optimizations (IPO) report
- Profile-Guided Optimizations (PGO) report
- High-level Optimizations (HLO) report
- Software-pipelining (SWP) report (IA-64 architecture only)
- Vectorization report
- Parallelism report
- OpenMP* report

Compiler Reports Quick Reference

The Intel® compiler provides the following options to generate and manage optimization reports:

Linux* and Mac OS* X	Windows*	Description
-opt-report or -opt-report N	/Qopt-report or /Qopt-report:N	Generates optimization report, with different levels of detail, directed to stderr. Valid values for N are 0 through 3. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail.
-opt-report-phase	/Qopt-report-phase	Specifies the optimization phase to use when generating reports. If you do not specify a phase the compiler defaults to all, which can adversely affect compile times. See Generating Reports for more information about the supported phases.
-opt-report-file	/Qopt-report-file	Generates an optimization report and directs the report output to the specified file name. If the file is not in the local directory, supply the full path to the output file. This option overrides the opt-report option.
-opt-report-routine	/Qopt-report-routine	Generates reports from all routines with names containing a string as part of their name; pass the string as an argument to this option. If not specified, the compiler will generate reports on all routines.

Linux* and Mac OS* X	Windows*	Description
-diag-<type>	/Qdiag-<type>	This option is not a compiler reports option; however, it controls the diagnostic message levels generated by the compiler, and it offers a means to provide useful information without needing to generate reports. <*type*> is a placeholder for several different values or lists.

This quick reference does not list the options for the vectorization, parallelism, or OpenMP* reports.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

If you use interprocedural optimization (IPO) options, you can request compiler reports when using the xi* tools , as described in Requesting Compiler Reports with the xi* Tools.

Generating Reports

Use the options listed in this topic to generate reports on the following optimizers.

- Interprocedural Optimization (IPO)
- Profile-guided Optimization (PGO)
- High Performance Optimizer (HPO)
- High-level Optimization (HLO)
- Intermediate Language Scalar Optimization (ILO)
- Software Pipelining (SWP)

Specify an optimizer phase by passing the `phase` argument to the `-opt-report-phase` (Linux* and Mac OS* X) or `/Qopt-report-phase` (Windows*) option.

Optimizer Phase	Supported Optimizer
`pgo`	Profile-guided Optimizer
`ipo`	Interprocedural Optimizer
`ilo`	Intermediate Language Scalar Optimizer

Optimizer Phase	Supported Optimizer
hpo	High Performance Optimizer
hlo	High-level Optimizer
ecg	Itanium® Compiler Code Generator Mac OS X: This phase is not supported.
ecg_swp	The software pipelining component of the Code Generator phase (Windows and Linux systems using IA-64 architecture only)
all	All optimizers supported on the architecture. This is not recommended; the resulting output can be too extensive to be useful. Experiment with targeted phase reports first.

Reports Available by Architecture and Option

IA-32, Intel® 64, and IA-64 architectures:

- ilo and pgo
- hpo and hlo: For IA-32 architecture, supported with -x (Linux and Mac OS X) or /Qx (Windows) option.

  For Intel® 64 architecture, supported with -O2 (Linux and Mac OS X) or /O2 (Windows) option. For IA-32 and Intel® 64 architectures, a subset of these optimizations are enabled at default optimization level (O2). For IA-64 architecture, supported with -O3 (Linux and Mac OS X) or /O3 (Windows) option.
- ipo: Interprocedural optimization is enabled for -O2 (Linux and Mac OS X) or /O2 (Windows) option or above.
- all: All of the above.

IA-64 architecture only:

- ecg

Running the Reports

Use syntax similar to the following to run the compiler reports.

Operating System	Sample Syntax
Linux and Mac OS X	ifort -c -opt-report 2 -opt-report-phase=all sample.f90
Windows	ifort /c /Qopt-report:2 /Qopt-report-phase:all sample.f90

The sample command instructs the compiler to generate a report and send the results to `stderr` and specifies the reports should include information about all available optimizers. In most cases, specifying `all` as the phase will generate too much information to be useful.

If you want to capture the report in an output file instead of sending it to `stderr`, specify `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) and indicate an output file name. If you do not want the compiler to invoke the linker, specify `-c` (Linux and Mac OS X) or `/c` (Windows) >as shown in the sample syntax above; by specifying the option you instruct the compiler to stop after generating object code and reporting the results.

See Compiler Reports Quick Reference for information about how to use the report related options.

When you specify a phase name, as shown above, the compiler generates all reports from that optimizer. The option can be used multiple times on the same command line to generate reports for multiple optimizers. For example, for if you specified `-opt-report-phase ipo -opt-report-phase hlo` (Linux and Mac OS X) or `/Qopt-report-phase ipo /Qopt-report-phase hlo` (Windows) the compiler generates reports from the interprocedural optimizer and the high-level optimizer code generator.

You do not need to fully specify an optimizer name in the command; in many cases, the first few characters should suffice to generate reports; however, all optimization reports that have a matching prefix are generated.

Each of the optimizer logical names supports many specific, targeted optimizations within them. Each of the targeted optimizations have the prefix of the optimizer name. Enter `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows) to list the names of optimizers that are supported. The following table lists some examples:

Optimizer	Description
`ipo_inl`	Interprocedural Optimizer, inline expansion of functions
`ipo_cp`	Interprocedural Optimizer, constant propagation
`hlo_unroll`	High-level Optimizer, loop unrolling

Optimizer	Description
hlo_prefetch	High-level Optimizer, prefetching

Viewing the Optimization Reports Graphically (Linux)

To generate the graphical report display, you must compile the application using the following optimization reports options, at a minimum: `-opt-report-phase` and `-opt-report-file`.

As with the text-based reports, the graphical report information can be generated on all architectures.

Interprocedural Optimizations (IPO) Report

The IPO report provides information on the functions that have been inlined and can help to identify the problem loops. The report can help to identify how and where the compiler applied IPO to the source files.

The following command examples demonstrate how to run the IPO reports with the minimum output.

Operating System	Syntax Examples
Linux* and Mac OS* X	`ifort -opt-report 1 -opt-report-phase=ipo a.f90 b.f90`
Windows*	`ifort /Qopt-report:1 /Qopt-report-phase:ipo a.f90 b.f90`

where `-opt-report` (Linux and Mac OS X) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase=ipo` (Linux and Mac OS X) or `/Qopt-report-phase:ipo` (Windows) indicates the phase (`ipo`) to report.

You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the Output

The IPO report details information in two general sections: whole program analysis and inlining. By default, the report generates a medium level of detail. You can specify an output file to capture the report results. Running maximum IPO report results can be very extensive and technical; specifying a file to capture the results can help to reduce analysis time. The following sample report illustrates the general layout.

Sample IPO Report

```
<;-1:-1;IPO;;0>
WHOLE PROGRAM (SAFE) [EITHER METHOD]: TRUE
WHOLE PROGRAM (SEEN) [TABLE METHOD]: TRUE
WHOLE PROGRAM (READ) [OBJECT READER METHOD]: TRUE
INLINING OPTION VALUES:
  -inline-factor: 100
  -inline-min-size: 20
  -inline-max-size: 230
  -inline-max-total-size: 2000
  -inline-max-per-routine: disabled
  -inline-max-per-compile: disabled
<ipo_sample_main.f90;29:37;IPO INLINING;MAIN__;0>
INLINING REPORT: (MAIN__) [1/4=25.0%]
  -> for_write_seq_lis(EXTERN)
  -> INLINE: mysum_(5) (isz = 23) (sz = 30 (13+17))
...
<ipo_sample_init.f90;32:40;IPO CONSTANT PROPAGATION;init_;0>
  CONSTANT PROPAGATION:  VARS(1)
...
```

The following table summarizes the common report elements and provides a general description to help interpret the results.

Report Element	Description
WHOLE PROGRAM (SAFE) [EITHER METHOD]:	TRUE or FALSE. • TRUE: The compiler determined, using one or both of the whole program analysis models, that the whole program was present during compilation. • FALSE: The compiler determined the whole program was not present during compilation. The compiler could not apply whole program IPO. See Inteprocedural Optimizations (IPO) Overview for more information on the whole program analysis models.
WHOLE PROGRAM (SEEN) [TABLE METHOD]:	TRUE or FALSE. • TRUE: The compiler resolved all references, either within the application code or in the standard table for the functions within the compiler. • FALSE: The compiler could not resolve all references. One or more functions references could not be found either in the user code or standard functions table.
WHOLE PROGRAM (READ) [OBJECT READER METHOD]:	TRUE or FALSE. • TRUE: The compiler determined that all conditions were met for linking at a level equivalent to the -O0 (Linux and Mac OS X) or /Od (Windows) option. • FALSE: The compiler could not resolve one or more references. The linking step failed.

Report Element	Description
INLINING OPTION VALUES:	Displays the compilation values used for the following developer-directed inline expansion options: • inline-factor • inline-min-size • inline-max-size • inline-max-total-size • inline-max-per-routine • inline-max-per-compile If you specify the one or more of the appropriate options, the report lists the values you specified; if you do not specify an option and value the compiler uses the defaults values for the listed options, and the compiler will list the default values. The values indicate the same intermediate language units listed in Compiler Options for each of these options. See Developer Directed Expansion of User Functions for more information about using these options.
INLINING REPORT:	Includes a string in the format of the following (<name>) [<current number>/<total number>=<percent complete>] where • <name>: the name of the function being reported on. • <current number> is the number of the function being reported on. Not every function can be inlined; gaps in the current number are common.

Report Element	Description
INLINE:	• <total number> is the total number of functions being evaluated. • <percent complete> is a simple percentage of the functions being inlined. If a function is inlined, the function line has the prefix "-> INLINE: _". The option reports displays the names of the functions. The report uses the following general syntax format: -> INLINE: _<name>(#) (isz) (sz) where • <name> : Indicates the name of the function Static functions display the function name with numbered suffix. • #: Indicates the unique integer specifying the function number. • sz: Indicates the function size before optimization. This is a rough estimate loosely representative of the number of original instructions before optimization. • isz: Indicates the function size after optimization. This value (isz) will always be less than or equal to the unoptimized size (sz). • exec_cnt: Indicates that Profile-guided Optimization was specified during compilation. Indicates the number of times the function was called from this site.

Report Element	Description
	NOTE. You can make the mangled names readable by entering the following: Linux: `echo <mangled_name> | c++filt` Windows: `undname <mangled_name>` Function calls that could not be inlined lack the `INLINE` prefix. Additionally, the compiler marks non-inlined functions using the following conventions: • `EXTERN` indicates the function contained an external function call for which the compiler was not supplied the code. • `ARGS_IN_REGS` indicates inlining did not occur. For IA-32 and Intel® 64 architectures, an alternative for functions that were not inlined is to allow the compiler to pass the function arguments in registers rather than using standard calling conventions; however, for IA-64 architecture this is the default behavior.
`DEAD STATIC FUNCTION ELIMINATION:`	Indicates the reported function is a dead static. Code does not need to be created for these functions. This behavior allows the compiler to reduce the overall code size.

Profile-guided Optimization (PGO) Report

The PGO report can help to identify can help identify where and how the compiler used profile information to optimize the source code. The PGO report is most useful when combined with the PGO compilation steps outlined in Profile an Application. Without the profiling data generated during the application profiling process the report will generally not provide useful information.

Combine the final PGO step with the reporting options by including `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*). The following syntax examples demonstrate how to run the report using the combined options.

Operating System	Syntax Examples
Linux and Mac OS X	`ifort -prof-use -opt-report -opt-report-phase=pgo pgotools_sample.f90`
Windows	`ifort /Qprof-use /Qopt-report /Qopt-report-phase:pgo pgotools_sample.f90`

By default the PGO report generates a medium level of detail. You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the Output

Running maximum PGO report results can produce long and detailed results. Depending on the sources being profiled, analyzing the report could be very time consuming. The following sample report illustrates typical results and element formatting for the default output.

Sample PGO Report

```
<pgotools_sample.f90;-1:-1;PGO;_DELEGATE.;0>
  DYN-VAL:  pgotools_sample.f90  _DELEGATE.
<pgotools_sample.f90;-1:-1;PGO;_ADDERMOD.;0>
  NO-DYN:   pgotools_sample.f90  _ADDERMOD.
<pgotools_sample.f90;-1:-1;PGO;_ADDERMOD_mp_MOD_ADD;0>
  DYN-VAL:  pgotools_sample.f90  _ADDERMOD_mp_MOD_ADD
<pgotools_sample.f90;-1:-1;PGO;_DELEGATE;0>
  DYN-VAL:  pgotools_sample.f90  _DELEGATE
<pgotools_sample.f90;-1:-1;PGO;_MAIN__;0>
  DYN-VAL:  pgotools_sample.f90  _MAIN__
<pgotools_sample.f90;-1:-1;PGO;_MAIN._MAIN_ADD;0>
  DYN-VAL:  pgotools_sample.f90  _MAIN._MAIN_ADD
<pgotools_sample.f90;-1:-1;PGO;;0>
      5 FUNCTIONS HAD VALID DYNAMIC PROFILES
      1 FUNCTIONS HAD NO DYNAMIC PROFILES
  FILE CURRENT  QUALITY METRIC:  91.7%
  FILE POSSIBLE QUALITY METRIC:  91.7%
  FILE QUALITY  METRIC RATIO:   100.0%
```

The following table summarizes some of the common report elements and provides a general description to help interpret the results.

Report Element	Description
String listing information about the function being reported on. The string uses the following format. <*source name*>;<*start line*>;<*end line*>;<*optimization*>; <*function name*>;<*element type*>	The compact string contains the following information: • <*source name*>: Name of the source file being examined. • <*start line*>: Indicates the starting line number for the function being examined. A value of -1 means that the report applies to the entire function. • <*end line*>: Indicates the ending line number for the function being examined. • <*optimization*>: Indicates the optimization phase; for this report the indicated phase should be PGO. • <*function name*>: Indicates the name of the function. • <*element type*>: Indicates the type of the report element; 0 indicates the element is a comment.
DYN-VAL	Indicates that valid profiling data was generated for the function indicated; the source file containing the function is also listed.
NO-DYN	Indicates that no profiling data was generated for the function indicated; the source file containing the function is also listed.
FUNCTIONS HAD VALID DYNAMIC PROFILES	Indicates the number of functions that had valid profile information.

Report Element	Description
FUNCTIONS HAD NO DYNAMIC PROFILES	Indicated the number of functions that did not have valid profile information. This element could indicate that the function was not executed during the instrumented executable runs.
FUNCTIONS HAD VALID STATIC PROFILES	Indicates the number of functions for which static profiles were generated. The most likely cause for having a non-zero number is that dynamic profiling did not happen and static profiles were generated for all of the functions.
IPO CURRENT QUALITY METRIC	Indicates the general quality, represented as a percentage value between 50% and 100%t. A value of 50% means no functions had dynamic profiles, and a value of 100% means that all functions have dynamic profiles. The larger the number the greater the percentage of functions that had dynamic profiles.
IPO POSSIBLE QUALITY METRIC	Indicates the number of possible dynamic profiles. This number represent the best possible value, as a percentage, for `Current Quality`. This number is the highest value possible and represents the ideal quality for the given data set and the instrumented executable.
IPO QUALITY METRIC RATIO	Indicates the ratio of `Possible Quality` to `Current Quality`. A value of 100% indicates that all dynamic profiles were accepted. Any value less than 100% indicates rejected profiles.

High-level Optimization (HLO) Report

High-level Optimization (HLO) performs specific optimizations based on the usefulness and applicability of each optimization. The HLO report can provide information on all relevant areas plus structure splitting and loop-carried scalar replacement, and it can provide information about interchanges not performed for the following reasons:

- Function call are inside the loop
- Imperfect loop nesting
- Reliance on data dependencies; dependencies preventing interchange are also reported.
- Original order was proper but it might have been considered inefficient to perform the interchange.

For example, the report can provide clues to why the compiler was unable to apply loop interchange to a loop nest that might have been considered a candidate for optimization. If the reported problems (bottlenecks) can be removed by changing the source code, the report suggests the possible loop interchanges.

Depending on the operating system, you must specify the following options to enable HLO and generate the reports:

- Linux* and Mac OS* X: `-x`, `-O2` or `-O3`, `-opt-report 3`, `-opt-report-phase=hlo`
- Windows*: `/Qx`, `/O2` or `/O3`, `/Qopt-report:3`, `/Qopt-report-phase:hlo`

See High-level Optimization Overview for information about enabling HLO.

The following command examples illustrate the general command needed to create HLO report with combined options.

Operating System	Example Command
Linux and Mac OS X	`ifort -c -xSSE3 -O3 -opt-report 3 -opt-report-phase=hlo sample.f90`
Windows	`ifort /c /QxSSE3 /O3 /Qopt-report:3 /Qopt-report-phase:hlo sample.f90`

You can use `-opt-report-file` (Linux and Mac OS X) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Reading the report results

The report provides information using a specific format. The report format for Windows* is different from the format on Linux* and Mac OS* X. While there are some common elements in the report output, the best way to understand what kinds of advice the report can provide is to show example code and the corresponding report output.

Example 1: This example illustrates the condition where a function call is inside a loop.

Example 1

```
subroutine foo (A, B, bound)
   integer i,j,n,bound
   integer A(bound), B(bound,bound)
   n = bound
   do j = 1, n
      do i = 1, n
        B(j,i) = B(j,i) + A(j)
        call bar(A,B)
      end do
   end do
   return
end subroutine foo
```

Regardless of the operating system, the reports list optimization results on specific functions by presenting a line above there reported action. The line format and description are included below.

The following table summarizes the common report elements and provides a general description to help interpret the results.

Report Element	Description
String listing information about the function being reported on. The string uses the following format.	The compact string contains the following information: • <*source name*>: Name of the source file being examined.

Report Element	Description
<source name>;<start line>;<end line>;<optimization>; <function name>;<element type> For example, the reports listed below report the following information: Linux and Mac OS X: <sample1.f90;-1:-1;hlo;foo_;0> Windows: <sample1.f90;-1:-1;hlo;_FOO;0>	• <*start line*>: Indicates the starting line number for the function being examined. A value of -1 means that the report applies to the entire function. • <*end line*>: Indicates the ending line number for the function being examined. • <*optimization*>: Indicates the optimization phase; for this report the indicated phase should be hlo. • <*function name*>: Name of the function being examined. • <*element type*>: Indicates the type of the report element; 0 indicates the element is a comment.

Several report elements grouped together.
`QLOOPS 2/2 ENODE LOOPS 2`
`unknown 0 multi_exit_do 0 do 2`
`linear_do 2`
`LINEAR HLO EXPRESSIONS: 17 / 18`
	Windows only: This section of the report lists the following information:
	• `QLOOPS`: Indicates the number of well-formed loops found out of the loops discovered.
	• `ENODE LOOPS`: Indicates number of preferred forms (canonical) of the loops generated by HLO. This indicates the number of loops generated by HLO.
	• `unknown`: Indicates the number of loops that could not be counted.
	• `multi_exit_do`: Indicates the countable loops containing multiple exits.
	• `do`: Indicates the total number of loops with trip counts that can be counted.
	• `linear_do`: Indicates the number of loops with bounds that can be represented in a linear form.

Report Element	Description
	• LINEAR HLO EXPRESSIONS: Indicates the number of expressions (first number) in all of the intermediate forms (ENODE) of the expression (second number) that can be represented in a linear form.

The code sample list above will result in a report output similar to the following.

Operating System	Example 1 Report Output
Linux and Mac OS X	<sample1.f90;-1:-1;hlo;foo_;0> High Level Optimizer Report (foo_) Block, Unroll, Jam Report: (loop line numbers, unroll factors and type of transformation) <sample1.f90;7:7;hlo_unroll;foo_;0> Loop at line 7 unrolled with remainder by 2
Windows	<sample1.f90;-1:-1;hlo;_FOO;0> High Level Optimizer Report (_FOO) QLOOPS 2/2 ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do 2 LINEAR HLO EXPRESSIONS: 17 / 18 _____ C:\samples\sample1.f90;6:6;hlo_linear_trans;_FOO;0> Loop Interchange not done due to: User Function Inside Loop Nest Advice: Loop Interchange, if possible, might help Loopnest at lines: 6 7 : Suggested Permutation: (1 2) --> (2 1)

Example 2: This example illustrates the condition where the loop nesting prohibits interchange.

Example 2

```
subroutine foo (A, B, bound)
   integer i,j,n,bound
   integer A(bound), B(bound,bound)
   n = bound
   do j = 1, n
```

Example 2

```
      A(j) = j + B(1,j)
      do i = 1, n
        B(j,i) = B(j,i) + A(j)
      end do
   end do
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 2 Report Output
Linux and Mac OS X	<sample2.f90;-1:-1;hlo;foo_;0> High Level Optimizer Report (foo_) Block, Unroll, Jam Report: (loop line numbers, unroll factors and type of transformation) <sample2.f90;8:8;hlo_unroll;foo_;0> Loop at line 8 unrolled with remainder by 2
Windows	<sample2.f90;-1:-1;hlo;_FOO;0> High Level Optimizer Report (_FOO) QLOOPS 2/2 ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do 2 LINEAR HLO EXPRESSIONS: 24 / 24 --- C:\samples\sample2.f90;6:6;hlo_linear_trans;_FOO;0> Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due t o other Compiler Transformations) Advice: Loop Interchange, if possible, might help Loopnest at lines: 6 8 : Suggested Permutation: (1 2) --> (2 1)

Example 3: This example illustrates the condition where data dependence prohibits loop interchange.

Example 3

```
subroutine foo (bound)
   integer i,j,n,bound
   integer A(100,100), B(100,100), C(100,100)
```

Example 3

```
   equivalence (B(2),A)
   n = bound
   do j = 1, n
      do i = 1, n
        A(j,i) = C(j,i) * 2
        B(j,i) = B(j,i) + A(j,i) * C(j,i)
      end do
   end do
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 3 Report Output
Linux and Mac OS X	<sample3.f90;-1:-1;hlo;foo_;0> High Level Optimizer Report (foo_) <sample3.f90;8:8;hlo_scalar_replacement;in foo_;0> #of Array Refs Scalar Replaced in foo_ at line 8=2 Block, Unroll, Jam Report: (loop line numbers, unroll factors and type of transformation) <sample3.f90;8:8;hlo_unroll;foo_;0> Loop at line 8 unrolled with remainder by 2
Windows	<sample3.f90;-1:-1;hlo;_FOO;0> High Level Optimizer Report (_FOO) QLOOPS 2/2 ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do 2 LINEAR HLO EXPRESSIONS: 24 / 24 ——————————————————————— C:\samples\sample3.f90;8:8;hlo_scalar_replacement;in _FOO ;0> #of Array Refs Scalar Replaced in _FOO at line 8=1 C:\samples\3.f90;7:7;hlo_linear_trans;_FOO;0> Loop Interchange not done due to: Data Dependencies Dependencies found between following statements: [From_Line# -> (Dependency Type) To_Line#] [9 ->(Flow) 10] [9 ->(Output) 10] [10

Operating System	Example 3 Report Output
	->(Anti) 10] [10 ->(Anti) 9] [10 ->(Output) 9] Advice: Loop Interchange, if possible, might help Loopnest at lines: 7 8 : Suggested Permutation: (1 2) --> (2 1)

Example 4: This example illustrates the condition where the loop order was determined to be proper, but loop interchange might offer only marginal relative improvement.

Example 4

```
subroutine foo (A, B, bound, value)
   integer i,j,n,bound,value
   integer A(bound, bound), B(bound,bound)
   n = bound
   do j = 1, n
      do i = 1, n
        A(i,j) = A(i,j) + B(j,i)
      end do
   end do
   value = A(1,1)
   return
end subroutine foo
```

The code sample listed above will result in a report output similar to the following.

Operating System	Example 4 Report Output
Linux and Mac OS X	<sample4.f90;-1:-1;hlo;foo_;0> High Level Optimizer Report (foo_) Block, Unroll, Jam Report: (loop line numbers, unroll factors and type of transformation) <sample4.f90;7:7;hlo_unroll;foo_;0> Loop at line 7 unrolled with remainder by 2
Windows	<sample4.f90;-1:-1;hlo;_FOO;0> High Level Optimizer Report (_FOO) QLOOPS 2/2 ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do 2 LINEAR HLO EXPRESSIONS: 18 / 18

Example 5: This example illustrates the conditions where the loop nesting was imperfect and the loop order was good, but loop interchange would offer marginal relative improvements.

Example

```
subroutine foo (A, B, C, bound, value)
   integer i,j,n,bound,value
   integer A(bound, bound), B(bound,bound), C(bound, bound)
   n = bound
   do j = 1, n
      value = value + A(1,1)
      do i = 1, n
        value = B(i,j) + C(j,i)
      end do
   end do
   return
```

Example
`end subroutine foo`

The code sample listed above will result in a report output similar to the following.

Operating System	Example 5 Report Output
Linux and Mac OS X	<sample5.f90;-1:-1;hlo;foo_;0> High Level Optimizer Report (foo_) Loopnest Preprocessing Report: <sample5.f90;7:8;hlo;foo_;0> Preprocess Loopnests <foo_>: Moving Out Store @Line<8> in Loop @Line<7>
Windows	<sample5.f90;-1:-1;hlo;_FOO;0> High Level Optimizer Report (_FOO) QLOOPS 2/2 ENODE LOOPS 2 unknown 0 multi_exit_do 0 do 2 linear_do 2 LINEAR HLO EXPRESSIONS: 20 / 25 --- Loopnest Preprocessing Report: C:\samples\sample5.f90;7:8;hlo;_FOO;0> Preprocess Loopnests <_FOO>: Moving Out Store @Line<8> in Loop @Line<7> C:\samples\sample5.f90;5:5;hlo_linear_trans;_FOO;0> Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due t o other Compiler Transformations) Advice: Loop Interchange, if possible, might help Loopnest at lines: 5 7 : Suggested Permutation: (1 2) --> (2 1)

Changing Code Based on the Report Results

While the HLO report tells you what loop transformations the compiler performed and provides some advice, the omission of a given loop transformation might imply that there are transformations the compiler might attempt. The following list suggests some transformations you might want to apply. (Manual optimization techniques, like manual cache blocking, should

be avoided or used only as a last resort.)

- Loop Interchanging - Swap the execution order of two nested loops to gain a cache locality or unit-stride access performance advantage.
- Distributing - Distribute or split up one large loop into two smaller loops. This strategy might provide an advantage when too many registers are being consumed in a large loop.
- Fusing - Fuse two smaller loops with the same trip count together to improve data locality.
- Loop Blocking - Use cache blocking to arrange a loop so it will perform as many computations as possible on data already residing in cache. (The next block of data is not read into cache until computations using the first block are finished.)
- Unrolling - Unrolling is a way of partially disassembling a loop structure so that fewer numbers of iterations of the loop are required; however, each resulting loop iteration is larger. Unrolling can be used to hide instruction and data latencies, to take advantage of floating point loadpair instructions, and to increase the ratio of real work done per memory operation.
- Prefetching - Request the compiler to bring data in from relatively slow memory to a faster cache several loop iterations ahead of when the data is actually needed.
- Load Pairing - Use an instruction to bring two floating point data elements in from memory in a single step.

High Performance Optimizer (HPO) Report

The following command examples illustrate the general command needed to create HPO report with combined options.

Operating System	Example Command
Linux* and Mac OS* X	`ifort -opt-report 0 -opt-report-phasehpo sample.f90`
Windows*	`ifort /Qopt-report:0 /Qopt-report-phase:hpo sample.f90`

Use `-opt-report-help` (Linux and Mac OS X) or `/Qopt-report-help` (Windows) to list the names of HPO report categories.

You must specify different compiler options to use the specific HPO report categories.

- For OpenMP*, add `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) to the command line.

- For parallelism, add `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows) to the command line.
- For vectorization, add `-x` (Linux and Mac OS X) or `/Qx` (Windows) to the command line; valid processor values are `SSE4.1`, `SSSE3`, `SSE3`, and `SSE2`.

Parallelism Report

The `-par-report` (Linux* and Mac OS* X) or `/Qpar-report` (Windows*) option controls the diagnostic levels 0, 1, 2, or 3 of the auto-parallelizer. Specify a value of 3 to generate the maximum diagnostic details.

Run the diagnostics report by entering commands similar to the following:

Operating System	Commands
Linux and Mac OS X	`ifort -c -parallel -par-report 3 sample.f90`
Windows	`ifort /c /Qparallel /Qpar-report:3 sample.f90`

where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the example without generating an executable.

NOTE. Linux and Mac OS X: The space between the option and the phase is optional. Windows: The colon between the option and phase is optional.

For example, assume you want a full diagnostic report on the following example code:

Example

```
subroutine no_par(a, MAX)
   integer :: i, a(MAX)
   do i = 1, MAX
     a(i) = mod((i * 2), i) * 1 + sqrt(3.0)
     a(i) = a(i-1) + i
   end do
end subroutine no_par
```

The following example output illustrates the diagnostic report generated by the compiler for the example code shown above. In most cases, the comment listed next to the line is self-explanatory.

Example Report Output

`procedure: NO_PAR`
 	
`sample.f90(7):(4) remark #15049: loop was not parallelized: loop is not a parallelization candidate`
 	
`sample.f90(7):(4) remark #15050: loop was not parallelized: existence of parallel dependence`
 	
`sample.f90(13):(6) remark #15051: parallel dependence: proven FLOW dependence between A line 13, and A line 13`

Responding to the results

The `-par-threshold{n)` (Linux* and Mac OS* X) or `/Qpar-threshold[:n]` (Windows*) option sets a threshold for auto-parallelization of loops based on the probability of profitable execution of the loop in parallel. The value of *n* can be from 0 to 100. You can use `-par-threshold0` (Linux and Mac OS X) or `/Qpar-threshold:0` (Windows) to auto-parallelize loops regardless of computational work.

Use `-ipo[value]` (Linux and Mac OS X) or `/Qipo` (Windows) to eliminate assumed side-effects done to function calls.

Use the `!DEC$ PARALLEL` directive to eliminate assumed data dependency.

Software Pipelining (SWP) Report (Linux* and Windows*)

The SWP report can provide details information about loops currently taking advantage of software pipelining available on IA-64 architecture based systems. The report can suggest reasons why the loops are not being pipelined.

The following command syntax examples demonstrates how to generate a SWP report for the Itanium® Compiler Code Generator (ECG) Software Pipeliner (SWP).

Operating System	Syntax Examples
Linux*	`ifort -c -opt-report -opt-report-phase=ecg_swp sample.f90`

Operating System	Syntax Examples
Windows*	`ifort /c /Qopt-report /Qopt-report-phase:ecg_swp sample.f90`

where `-c` (Linux) or `/c` (Windows) tells the compiler to stop at generating the object code (no linking occurs), `-opt-report` (Linux) or `/Qopt-report` (Windows) invokes the report generator, and `-opt-report-phase=ecg_swp` (Linux) or `/Qopt-report-phase:ecg_swp` (Windows) indicates the phase (ecg) for which to generate the report.

You can use `-opt-report-file` (Linux) or `/Qopt-report-file` (Windows) to specify an output file to capture the report results. Specifying a file to capture the results can help to reduce the time you spend analyzing the results and can provide a baseline for future testing.

Typically, loops that software pipeline will have a line that indicates the compiler has scheduled the loop for SWP in the report. If the `-O3` (Linux) or `/O3` (Windows) option is specified, the SWP report merges the loop transformation summary performed by the loop optimizer.

Some loops will not software pipeline (SWP) and others will not vectorize if function calls are embedded inside your loops. One way to get these loops to SWP or to vectorize is to inline the functions using IPO.

You can compile this example code to generate a sample SWP report. The sample reports is also shown below.

Example

```
!#define NUM 1024
subroutine multiply_d(a,b,c,NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
  NUM=1024
  do i=0,NUM
    do j=0,NUM
      do k=0,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine multiply_d
```

The following sample report shows the report phase that results from compiling the example code shown above (when using the `ecg_swp` phase).

Sample SWP Report

```
Swp report for loop at line 10 in _Z10multiply_dPA1024_dS0_S0_ in file SWP report.f90
 Resource II  = 2
 Recurrence II = 2
 Minimum II  = 2
 Scheduled II = 2

 Estimated GCS II   = 7
```

Sample SWP Report

```
Percent of Resource II needed by arithmetic ops  = 100%
Percent of Resource II needed by memory ops  =  50%
Percent of Resource II needed by floating point ops =  50%
Number of stages in the software pipeline = 6
```

Reading the Reports

To understand the SWP report results, you must know something about the terminology used and the related concepts. The following table describes some of the terminology used in the SWP report.

Term	Definition
II	Initiation Interval (II). The number of cycles between the start of one iteration and the next in the SWP. The presence of the term II in any SWP report indicates that SWP succeeded for the loop in question. II can be used in a quick calculation to determine how many cycles your loop will take, if you also know the number of iterations. Total cycle time of the loop is approximately N * Scheduled II + number Stages (Where N is the number of iterations of the loop). This is an approximation because it does not take into account the ramp-up and ramp-down of the prolog and epilog of the SWP, and only considers the kernel of the SWP loop. As you modify your code, it is generally better to see scheduled II go down, though it is really N* (Scheduled II) + Number of stages in the software pipeline that is ultimately the figure of merit.

Term	Definition
Resource II	Resource II implies what the Initiation Interval should be when considering the number of functional units available.
Recurrence II	Recurrence II indicates what the Initiation Interval should be when there is a recurrence relationship in the loop. A recurrence relationship is a particular kind of a data dependency called a flow dependency like `a[i] = a[i-1]` where `a[i]` cannot be computed until `a[i-1]` is known. If Recurrence II is non-zero and there is no flow dependency in the code, then this indicates either Non-Unit Stride Access or memory aliasing. See Helping the Compiler for more information.
Minimum II	Minimum II is the theoretical minimum Initiation Interval that could be achieved.
Scheduled II	Scheduled II is what the compiler actually scheduled for the SWP.
number of stages	Indicates the number of stages. For example, in the report results below, the line "Number of stages in the software pipeline = 3" indicates there were three stages of work, which will show, in assembly, to be a load, an FMA instruction and a store.
loop-carried memory dependence edges	The loop-carried memory dependence edges means the compiler avoided WAR (Write After Read) dependency. Loop-carried memory dependence edges can indicate problems with memory aliasing. See Helping the Compiler.

Using the Report to Resolve Issues

One fast way to determine if specific loops have been software pipelined is to look for "r;Number of stages in the software pipeline" in the report; the phrase indicates that software pipelining for the associated loop was successfully applied.

Analyze the loops that did not SWP in order to determine how to enable SWP. If the compiler reports the "Loop was not SWP because...", see the following table for suggestions about how to correct possible problems:

Message in Report	Suggested Action
acyclic global scheduler can achieve a better schedule: => loop not pipelined	Indicates that the most likely cause is memory aliasing issues. For memory alias problems see memory aliasing (restrict, `#pragma ivdep`). Might indicate the application is accessing memory in a non-Unit Stride fashion. Non-Unit Stride issues may be indicated by an artificially high recurrence II; If you know there is no recurrence relationship (`a[i] = a[i-1] + b[i]`) in the loop, then a high recurrence II (greater than 0) is a sign that you are accessing memory non-Unit Stride. Rearranging code, perhaps a loop interchange, might help mitigate this problem.
Loop body has a function call	Indicates inlining the function might help solve the problem.
Not enough static registers	Indicates you should distribute the loop by separating it into two or more loops. On IA-64 architecture based systems you may use `#pragma distribute point`.
Not enough rotating registers	Indicates the loop carried values use the rotating registers. Distribute the loop.
Loop too large	Indicates you should distribute the loop.

Message in Report	Suggested Action
	On IA-64 architecture based systems you may use the `#pragma distribute point`.
Loop has a constant trip count < 4	Indicates unrolling was insufficient. Attempt to fully unroll the loop. However, with small loops fully unrolling the loop is not likely to affect performance significantly.
Too much flow control	Indicates complex loop structure. Attempt to simplify the loop.

Index variable type used can greatly impact performance. In some cases, using loop index variables of type short or unsigned int can prevent software pipelining. If the report indicates performance problems in loops where the index variable is not int and if there are no other obvious causes, try changing the loop index variable to type int.

Vectorization Report

The vectorization report can provide information about loops that could take advantage of Intel® Streaming SIMD Extensions (Intel® SSE3, SSE2, and SSE) vectorization, and it is available on systems based on IA-32 and Intel® 64 architectures.

See Using Parallelism for information on other vectorization options.

The `-vec-report` (Linux* and Mac OS* X) or `/Qvec-report` (Windows*) option directs the compiler to generate the vectorization reports with different levels of information. Specify a value of 3 to generate the maximum diagnostic details.

Operating System	Command
Linux and Mac OS X	`ifort -c -xSSSE3 -vec-report3 sample.f90`
Windows	`ifort /c /QxSSSE3 /Qvec-report:3 sample.f90`

where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the example without generating an executable.

NOTE. Linux and Mac OS X: The space between the option and the phase is optional.

Windows: The colon between the option and phase is optional.

The following example results illustrate the type of information generated by the vectorization report:

Example results

```
sample.f90(27) : (col. 9) remark: loop was not vectorized: not inner loop.

sample.f90(28) : (col. 11) remark: LOOP WAS VECTORIZED.

sample.f90(31) : (col. 9) remark: loop was not vectorized: not inner loop.

sample.f90(32) : (col. 11) remark: LOOP WAS VECTORIZED.

sample.f90(37) : (col. 10) remark: loop was not vectorized: not inner loop.

sample.f90(38) : (col. 12) remark: loop was not vectorized: not inner loop.

sample.f90(40) : (col. 14) remark: loop was not vectorized: vectorization possible but
seems inefficient.

sample.f90(46) : (col. 10) remark: loop was not vectorized: not inner loop.

sample.f90(47) : (col. 12) remark: loop was not vectorized: contains unvectorizable
statement at line 48.
```

If the compiler reports "r;Loop was not vectorized" because of the existence of vector dependence, then you should analyze the loop for vector dependence. If you determine there is no legitimate vector dependence, then the message indicates that the compiler was assuming the pointers or arrays in the loop were dependent, which implies the pointers or arrays were aliased. Use memory disambiguation techniques to resolve these cases.

There are three major types of vector dependence: `FLOW`, `ANTI`, and `OUTPUT`.

There are a number of situations where the vectorization report may indicate vector dependencies. The following situations will sometimes be reported as vector dependencies, non-unit stride, low trip count, and complex subscript expression.

Non-Unit Stride

The report might indicate that a loop could not be vectorized when the memory is accessed in a non-Unit Stride manner. This means that nonconsecutive memory locations are being accessed in the loop. In such cases, see if loop interchange can help or if it is practical. If not then you can force vectorization sometimes through `vector always` directive; however, you should verify improvement.

See Understanding Runtime Performance for more information about non-unit stride conditions.

Usage with Other Options

The vectorization reports are generated during the final compilation phase, which is when the executable is generated; therefore, there are certain option combinations you cannot use if you are attempting to generate a report. If you use the following option combinations, the compiler issues a warning and does not generate a report:

- `-c` or `-ipo` or `-x` with `-vec-report` (Linux* and Mac OS* X) and `/c` or `/Qipo` or `/Qx` with `/Qvec-report` (Windows*)
- `-c` or `-ax` with `-vec-report` (Linux and Mac OS X) and `/c` or `/Qax` with `/Qvec-report` (Windows)

The following example commands can generate vectorization reports:

Operating System	Command Examples
Linux and Mac OS X	`ifort -xSSSE3 -vec-report3 sample.f90` `ifort -xSSSE3 -ipo -vec-report3 sample.f90`
Windows	`ifort /QxSSSE3 /Qvec-report:3 sample.f90` `ifort /QxSSSE3 /Qipo /Qvec-report:3 sample.f90`

Responding to the Results

You might consider changing existing code to allow vectorization under the following conditions:

- The vectorization report indicates that the program "contains unvectorizable statement at line XXX".
- The vectorization report states there is a "vector dependence: proven FLOW dependence between 'r;variable' line XXX, and 'r;variable' line XXX" or "loop was not vectorized: existence of vector dependence." Generally, these conditions indicate true loop dependencies are stopping vectorization. In such cases, consider changing the loop algorithm.

For example, consider the two equivalent algorithms producing identical output below. "Foo" will not vectorize due to the FLOW dependence but "bar" does vectorize.

Example

```
subroutine foo(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y (i) = y (i-1)+1
  end do
end subroutine foo
subroutine bar(y)
  implicit none
  integer :: i
  real :: y(10)
  do i=2,10
    y (i) = y (1)+i
  end do
end subroutine bar
```

Unsupported loop structures may prevent vectorization. An example of an unsupported loop structure is a loop index variable that requires complex computation. Change the structure to remove function calls to loop limits and other excessive computation for loop limits.

Example

```
function func(n)
  implicit none
  integer :: func, n
  func = n*n-1
end function func
subroutine unsupported_loop_structure(y,n)
  implicit none
  integer :: i,n, func
  real :: y(n)
  do i=0,func(n)
    y(i) = y(i) * 2.0
  end do
end subroutine unsupported_loop_structure
```

Non-unit stride access might cause the report to state that "vectorization possible but seems inefficient". Try to restructure the loop to access the data in a unit-stride manner (for example, apply loop interchange), or try directive .

Using mixed data types in the body of a loop might prevent vectorization. In the case of mixed data types, the vectorization report might state something similar to "loop was not vectorized: condition too complex".

The following example code demonstrates a loop that cannot vectorize due to mixed data types within the loop. For example, `withinborder` is an integer while all other data types in loop are not. Simply changing the `withinborder` data type will allow this loop to vectorize.

Example

```
subroutine howmany_close(x,y,n)
  implicit none
  integer :: i,n,withinborder
  real :: x(n), y(n), dist
  withinborder=0
  do i=0,100
    dist=sqrt(x(i)*x(i) + y(i)*y(i))
    if (dist<5) withinborder= withinborder+1
  end do
end subroutine howmany_close
```

OpenMP* Report

The `-openmp-report` (Linux* and Mac OS* X) or `/Qopenmp-report` (Windows*) option controls the diagnostic levels for OpenMP* reporting. The OpenMP* report information is not generated unless you specify the option with a value of either 1 or 2. Specifying 2 provides the most useful information.

You must specify the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) along with this option.

Linux OS and Mac OS X	Windows OS	Description
`-openmp-report 2`	`/Qopenmp-report:2`	Report results are same as when specifying 1 except the results also include diagnostics indicating constructs, like directives, that were handled successfully. This is the recommend level.

Linux OS and Mac OS X	Windows OS	Description
`-openmp-report 1`	`/Qopenmp-report:1`	Reports on loops, regions, and sections that were parallelized successfully. This is the default level.
`-openmp-report 0`	`/Qopenmp-report:0`	No diagnostics report generated.

The following example commands demonstrate how to run the report using the combined commands.

Operating System	Syntax Examples
Linux and Mac OS X	`ifort -openmp -openmp-report 2 sample1.f90 sample2.f90`
Windows	`ifort /Qopenmp /Qopenmp-report:2 sample1.f90 sample2.f90`

NOTE. Linux and Mac OS X: The space between the option and the level is optional.

Windows: The colon between the option and level is optional.

The following example results illustrate the typical format of the generated information:

Example results
`openmp_sample.f90(77): (col. 7) remark : OpenMP DEFINED LOOP WAS PARALLELIZED.`
`openmp_sample.f90(68): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.`

See also:

- OpenMP* Options Quick Reference for information about these options.
- OpenMP* Support Overview for information on using OpenMP* in Intel® compilers.

25 Using Compiler Optimizations

Automatic Optimizations Overview

Intel® compilers allow you to compile applications for processors based on IA-32 architectures (32-bit applications), Intel® 64 architectures (32-bit and 64-bit applications), or IA-64 architectures (64-bit applications).

By default the compiler chooses a set of optimizations that balances compile-time and run-time performance for your application. Also, you can manually select optimizations based on the specific needs of the application.

The following table summarizes the common optimization options you can use for quick, effective results.

Linux* and Mac OS* X	Windows*	Description
`-O3`	`/O3`	Enables aggressive optimization for code speed. Recommended for code with loops that perform substantial calculations or process large data sets.
`-O2` (or `-O`)	`/O2`	Affects code speed. This is the default option; the compiler uses this optimization level if you do not specify anything.
`-O1`	`/O1`	Affects code size and locality. Disables specific optimizations.
`-fast`	`/fast`	Enables a collection of common, recommended optimizations for run-time performance. Can introduce architecture dependency.
`-O0`	`/Od`	Disables optimization. Use this for rapid compilation while debugging an application.

The variety of automatic optimizations enable you to quickly enhance your application performance. In addition to automatic optimizations, the compiler invokes other optimization enabled with source code directives, optimized library routines, and performance-enhancing utilities.

The remaining topics in this section provide more details on the automatic optimizations supported in the Intel compilers.

See Also

- Using Compiler Optimizations
- Enabling Automatic Optimizations
- Restricting Optimizations

Enabling Automatic Optimizations

This topic lists the most common code optimization options, describes the characteristics shared by IA-32, Intel® 64, and IA-64 architectures, and describes the general behavior for each architecture.

The architectural differences and compiler options enabled or disabled by these options are also listed in more specific detail in the associated Compiler Options topics; therefore, each option discussion listed below includes a link to the appropriate reference topic.

Linux* and Mac OS* X	Windows*	Description
-O1	/O1	Optimizes to favor smaller code size and code locality. In most cases, -O2 (Linux* OS and Mac OS* X) or /O2 (Windows* OS) is recommended over this option. This optimization disables some optimizations that normally increase code size. This level might improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops. In general, this optimization level does the following:

Linux* and Mac OS* X	Windows*	Description
		• Enables global optimization. • Disables intrinsic recognition and inlining of intrinsics. IA-64 architecture: • The option disables software pipelining, loop unrolling, and global code scheduling.
-O2 or -O	/O2	Optimizes for code speed. Since this is the default optimization, if you do not specify an optimization level the compiler will use this optimization level automatically. This is the generally recommended optimization level; however, specifying other compiler options can affect the optimization normally gained using this level. In general, the resulting code size will be larger than the code size generated using -O1 (Linux and Mac OS X) or /O1 (Windows). This option enables the following capabilities for performance gain: inlining intrinsic functions, constant propagation, copy propagation, dead-code elimination, global register allocation, global instruction scheduling and control speculation, loop unrolling,

Linux* and Mac OS* X	Windows*	Description
		optimized code selection, partial redundancy elimination, strength reduction/induction variable simplification, variable renaming, exception handling optimizations, tail recursions, peephole optimizations, structure assignment lowering optimizations, and dead store elimination. For IA-32 and Intel 64 architectures: • Enables certain optimizations for speed, such as vectorization. IA-64 architecture: • Enables optimizations for speed, including global code scheduling, software pipelining, predication, speculation, and data prefetch.
-O3	/O3	Enables -O2 (Linux and Mac OS X) or /O2 (Windows) optimizations, as well as more aggressive optimizations, including prefetching, scalar replacement, cache blocking, and loop and memory access transformations. As compared to -O2 (Linux) or /O2 (Windows), the optimizations enabled by this option often result in faster program execution, but can

Linux* and Mac OS* X	Windows*	Description
		slow down code execution in some cases. Using this option may result in longer compilation times. This option is recommended for loop-intensive applications that perform substantial floating-point calculations or process large data sets.
-fast	/fast	Provides a single, simple optimization that enables a collection of optimizations that favor run-time performance. This is a good, general option for increasing performance in many programs. For IA-32 and Intel 64 architectures, the -xSSSE3 (Linux and Mac OS X) or /QxSSSE3 (Windows) option that is set by this option cannot be overridden by other command line options. If you specify this option along with a different processor-specific option, such as -xSSE2 (Linux) or /QxSSE2 (Windows), the compiler will issue a warning stating the -xSSSE3 or /QxSSSE3 option cannot be overridden; the best strategy for dealing with this restriction is to explicitly specify the options you want to set from the command line.

Linux* and Mac OS* X	Windows*	Description
		CAUTION. Programs compiled with the -xSSSE3 (Linux and Mac OS X) or /QxSSSE3 (Windows) option will detect non-compatible processors and generate an error message during execution. While this option enables other options quickly, the specific options enabled by this option might change from one compiler release to the next. Be aware of this possible behavior change in the case where you use makefiles.

The following syntax examples demonstrate using the default option to compile an application:

Operating System	Example
Linux and Mac OS X	`ifort -O2 prog.f90`
Windows	`ifort /O2 prog.f90`

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Targeting IA-32 and Intel(R) 64 Architecture Processors Automatically

The -x (Linux* and Mac OS* X) or /Qx (Windows*) option can automatically optimize your application for specific Intel® processors based on IA-32 and Intel® 64 architectures.

The automatic optimizations allow you to take advantage of the architectural differences, new instruction sets, or advances in processor design; however, the resulting, optimized code might contain unconditional use of features that are not supported on other, earlier processors. Therefore, using these options effectively sets a minimum hardware requirement for your application.

The optimizations can include generating Intel® Streaming SIMD Extensions 4 (SSE4), Supplemental Streaming SIMD Extensions 3 (SSSE3), Streaming SIMD Extensions 3 (SSE3), Streaming SIMD Extensions 2 (SSE2), or Streaming SIMD Extensions (SSE) instructions.

If you intend to run your programs on multiple processors based on IA-32 or Intel® 64 architectures, do not use this option; instead, consider using the `-ax` (Linux and Mac OS X) or `/Qax` (Windows) option to achieve both processor-specific performance gains and portability among different processors.

Linux OS and Mac OS X	Windows OS	Description
`-xHost`	`/QxHost`	Can generate instructions for the highest instruction set and processor available on the compilation host.
`-xAVX`	`/QxAVX`	Optimizes for Intel processors that support Intel® Advanced Vector Extensions (Intel® AVX).
`-xSSE4.1`	`/QxSSE4.1`	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated.
`-xSSE4.2`	`/QxSSE4.2`	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE

Linux OS and Mac OS X	Windows OS	Description
		instructions and it can optimize for the Intel® Core™ processor family.
-xSSSE3	/QxSSSE3	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. This replaces value T, which is deprecated.
-xSSE3_ATOM	/QxSSE3_ATOM	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology. Can generate MOVBE instructions, depending on the setting of option -minstruction (Linux and Mac OS) or /Qinstruction (Windows). Mac OS X: Supported on IA-32 architectures.
-xSSE3	/QxSSE3	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. This replaces value P, which is deprecated. Mac OS X: Supported on IA-32 architectures.
-xSSE2	/QxSSE2	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium®

Linux OS and Mac OS X	Windows OS	Description
		4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. Mac OS X: Not supported.

Certain keywords for compiler options -m and /arch produce binaries that should run on processors not made by Intel that implement the same capabilities as the corresponding Intel processors. For details, see Compiler Options.

To prevent illegal instruction and similar unexpected run-time errors during program execution, the compiler inserts code in the main routine of the program to check for proper processor usage. Using this option limits you to a minimum processor level. For example, if you target an application to run on Intel® Xeon® processors based on the Intel® Core™ microarchitecture, it is unlikely the resulting application will operate correctly on earlier Intel processors.

If you target more than one processor value, the resulting code will be generated for the highest-performing processor specified if the compiler determines there is an advantage in doing so. The highest- to lowest-performing processor values are as follows:

1. SSE4.1
2. SSSE3
3. SSE3
4. SSE2

Executing programs compiled with processor values of SSE4.1, SSSE3, SSE3, or SSE2 on unsupported processors will display a run-time error. For example, if you specify the SSSE3 processor value to compile an application but execute the application on an Intel® Pentium® 4 processor, the application generates an error similar to the following:

Run-time Error

```
Fatal Error: This program was not built to run on the processor in your
system.

The allowed processors are: Intel(R) Core(TM) Duo processors and compatible
Intel processors with supplemental Streaming SIMD Extensions 3 (SSSE3)
instruction support.
```

The following examples demonstrate compiling an application for Intel® Core™2 Duo processor and compatible processors. The resulting binary might not execute correctly on earlier processors or on IA-32 architecture processors not made by Intel Corporation.

Operating System	Example
Linux and Mac OS X	`ifort -xSSSE3 sample.f90`
Windows	`ifort /QxSSSE3 sample.f90`

Targeting Multiple IA-32 and Intel(R) 64 Architecture Processors for Run-time Performance

The `-ax` (Linux* and Mac OS* X) or `/Qax` (Windows*) option instructs the compiler to determine if opportunities exist to generate multiple, specialized code paths to take advantage of performance gains and features available on newer Intel® processors based on IA-32 and Intel® 64 architectures. This option also instructs the compiler to generate a more generic (baseline) code path that should allow the same application to run on a larger number of processors; however, the baseline code path is usually slower than the specialized code.

The compiler inserts run-time checking code to help determine which version of the code to execute. The size of the compiled binary increases because it contains both a processor-specific version of some of the code and a generic baseline version of all code. Application performance is affected slightly due to the run-time checks needed to determine which code to use. The code path executed depends strictly on the processor detected at run time.

Processor support for the baseline code path is determined by the processor family or instruction set specified in the `-m` or `-x` (Linux and Mac OS X) or `/arch` or `/Qx` (Windows) option, which has default values for each architecture.

This allows you to impose a more strict processor or instruction set requirement for the baseline code path; however, such generic baseline code will not operate correctly on processors that are not compatible with the minimum processor or instruction set requirement. For the IA-32 architecture, you can specify a baseline code path that will work on all IA-32 compatible processors using the `-mia32` (Linux) or `/arch:IA32` (Windows) options. You should always specify the processor or instruction set requirements explicitly for the baseline code path, rather than depend on the defaults for the architecture.

Optimizations in the specialized code paths can include generating and using Intel® Streaming SIMD Extensions 4 (SSE4), Supplemental Streaming SIMD Extensions 3 (SSSE3), Streaming SIMD Extensions 3 (SSE3), or Streaming SIMD Extensions 2 (SSE2) instructions for supported Intel processors; however, such specialized code paths are executed only after checking verifies that the code is supported by the run-time host processor.

If not indicated otherwise, the following processor values are valid for IA-32 and Intel® 64 architectures.

Linux OS and Mac OS X	Windows OS	Description
-axSSE4.2	/QaxSSE4.2	Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core™ i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.
-axSSE4.1	/QaxSSE4.1	Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator instructions for Intel processors. Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for Intel® 45nm Hi-k next generation Intel® Core™ microarchitecture. This replaces value S, which is deprecated. Mac OS X: IA-32 and Intel® 64 architectures.
-axSSSE3	/QaxSSSE3	Can generate Intel® SSSE3, SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for the Intel® Core™2 Duo processor family. This replaces value T, which is deprecated. Mac OS X: IA-32 architecture.

Linux OS and Mac OS X	Windows OS	Description
-axSSE3_ATOM	/QaxSSE3_ATOM	Optimizes for the Intel® Atom™ processor and Intel® Centrino® Atom™ Processor Technology. Can generate MOVBE instructions, depending on the setting of option -minstruction (Linux and Mac OS) or /Qinstruction (Windows). Mac OS X: Supported on IA-32 architectures.
-axSSE3	/QaxSSE3	Can generate Intel® SSE3, SSE2, and SSE instructions for Intel processors and it can optimize for processors based on Intel® Core™ microarchitecture and Intel NetBurst® microarchitecture. This replaces value P, which is deprecated. Mac OS X: IA-32 architecture.
-axSSE2	/QaxSSE2	Can generate Intel® SSE2 and SSE instructions for Intel processors, and it can optimize for Intel® Pentium® 4 processors, Intel® Pentium® M processors, and Intel® Xeon® processors with Intel® SSE2. Linux and Windows: IA-32 architecture.

NOTE. You can specify -diag-disable cpu-dispatch (Linux and Mac OS X) or /Qdiag-disable:cpu-dispatch (Windows) to disable the display of remarks about multiple code paths for CPU dispatch.

If your application for IA-32 or Intel® 64 architectures does not need to run on multiple processors , consider using the `-x` (Linux and Mac OS X) or `/Qx` (Windows) option instead of this option.

The following compilation examples demonstrate how to generate an IA-32 architecture executable that includes an optimized version for Intel® Core™2 Duo processors, as long as there is a performance gain, an optimized version for Intel® Core™ Duo processors, as long as there is a performance gain, and a generic baseline version that runs on any IA-32 architecture processor.

NOTE. If you combine the arguments, you must add a comma (",") separator between the individual arguments.

Operating System	Example
Linux	`ifort -axSSSE3,SSE3 -mia32 sample.f90`
Windows	`ifort /QaxSSSE3,SSE3 /arch:IA32 sample.f90`

Targeting IA-64 Architecture Processors Automatically

The Intel compiler supports options that optimize application performance for Intel® Itanium® processors based on the IA-64 architecture.

Linux* OS	Windows* OS	Optimizes applications for...
`-mtune=itanium2-p9000`	`/G2-p9000`	Default. Dual-Core Intel® Itanium® 2 processor (9000 series)
`-mtune=itanium2`	`/G2`	Intel® Itanium® 2 processors

NOTE. Mac OS* X: These options are not supported.

While the resulting executable is backward compatible, generated code is optimized for specific processors; therefore, code generated with `-mtune=itanium2-p9000` (Linux) or `/G2-p9000` (Windows) will run correctly on Itanium® 2 processors.

The following examples demonstrate using the default options to target an Itanium® 2 processor (9000 series). The same binary will also run on Intel® Itanium® 2 processors.

Operating System	Example
Linux	`ifort -mtune=itanium2-p9000 prog.f90`
Windows	`ifort /G2-p9000 prog.f90`

Restricting Optimizations

The following table lists options that restrict the ability of the Intel® compiler to optimize programs.

Linux* and Mac OS* X	Windows*	Effect
`-O0`	`/Od`	Disables all optimizations. Use this during development stages where fast compile times are desired. Linux* and Mac OS* X: • Sets option `-fomit-frame-pointer` and option `-fmath-errno`. Windows*: • Use `/Od` to disable all optimizations while specifying particular optimizations, such as: `/Od /Ob1` (disables all optimizations, but only enables inlining) For more information, see the following topic: • `-O0` compiler option

Linux* and Mac OS* X	Windows*	Effect
-g	/Zi, /Z7	Generates symbolic debugging information in object files for use by debuggers. This option enables or disables other compiler options depending on architecture and operating system; for more information about the behavior, see the following topic: • -g compiler option
-fmath-errno, -fno-math-errno	No equivalent	Instructs the compiler to assume that the program tests errno after calls to math library functions. For more information, see the following topic: • -fmath-errno compiler option

Diagnostic Options

Linux and Mac OS X	Windows	Effect
-sox	/Qsox	Instructs the compiler to save the compiler options and version number in the executable. During the linking process, the linker places information strings into the resulting executable. Slightly increases file size, but using this option can make

Linux and Mac OS X	Windows	Effect
		identifying versions for regression issues much easier. For more information, see the following topic: • -sox compiler option

Using Parallelism: OpenMP* Support

26

OpenMP* Support Overview

The Intel® compiler supports the OpenMP* Version 3.0 API specification. For complete Fortran language support for OpenMP, see the OpenMP Application Program Interface Version 3.0 specification, which is available from the OpenMP web site (http://www.openmp.org/, click the Specifications link).

This version of the Intel compiler also introduces OpenMP API Version 3.0 API specification support, as described in the OpenMP web site (http://www.openmp.org/, click Specifications).

OpenMP provides symmetric multiprocessing (SMP) with the following major features:

- Relieves the user from having to deal with the low-level details of iteration space partitioning, data sharing, and thread creation, scheduling, and synchronization.
- Provides the benefit of the performance available from shared memory multiprocessor and multi-core processor systems on IA-32, Intel® 64, and IA-64 architectures, including those processors with Hyper-Threading Technology.

The compiler performs transformations to generate multithreaded code based on a developer's placement of OpenMP directives in the source program making it easy to add threading to existing software. The Intel compiler supports all of the current industry-standard OpenMP directives and compiles parallel programs annotated with OpenMP directives.

The compiler provides Intel-specific extensions to the OpenMP Version 3.0 specification including run-time library routines and environment variables. However, these extensions are only supported by the Intel compilers. A summary of the compiler options that apply to OpenMP* appears in OpenMP* Options Quick Reference.

Parallel Processing with OpenMP

To compile with OpenMP, you need to prepare your program by annotating the code with OpenMP directives in the form of the Fortran program comments. The Intel compiler processes the application and internally produces a multithreaded version of the code which is then compiled. The output is an executable with the parallelism implemented by threads that execute parallel regions or constructs. See Programming with OpenMP.

Using Other Compilers

The OpenMP specification does not define interoperability of multiple implementations; therefore, the OpenMP implementation supported by other compilers and OpenMP support in Intel compilers might not be interoperable. Even if you compile and build the entire application with one compiler, be aware that different compilers might not provide OpenMP source compatibility that would allow you to compile and link the same set of application sources with a different compiler and get the expected parallel execution results.

Intel compilers include two sets of OpenMP libraries, as described in OpenMP Source Compatibility and Interoperability with Other Compilers.

OpenMP* Options Quick Reference

These options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* OS and Mac OS* X	Windows* OS	Description
-openmp	/Qopenmp	This option enables the parallelizer to generate multi-threaded code based on the OpenMP* directives. The code can be executed in parallel on both uniprocessor and multiprocessor systems. IA-64 architecture only: • Implies -opt-mem-band-with1 (Linux) or /Qopt-mem-bandwidth1 (Windows).
-openmp-report	/Qopenmp-report	This option controls the OpenMP parallelizer's level of diagnostic messages. To use this option, you must also specify -openmp (Linux and Mac OS X) or /Qopenmp (Windows).

Linux* OS and Mac OS* X	Windows* OS	Description
-openmp-stubs	/Qopenmp-stubs	This option enables compilation of OpenMP programs in sequential mode. The OpenMP directives are ignored and a stub OpenMP library is linked.
-openmp-profile	/Qopenmp-profile	This option enables analysis of OpenMP* applications. To use this option, you must have previously installed Intel® Thread Profiler, which is one of the Intel® Threading Analysis Tools. This option can adversely affect performance because of the additional profiling and error checking invoked to enable compatibility with the threading tools. Do not use this option unless you plan to use the Intel® Thread Profiler.
-openmp-lib	/Qopenmp-lib	This option lets you specify an OpenMP* run-time library to use for linking. The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers. The compatibility OpenMP run-time library is compatible with object files created using the Microsoft* OpenMP run-time library (vcomp) and GNU OpenMP run-time library (libgomp).

Linux* OS and Mac OS* X	Windows* OS	Description
-openmp-link	/Qopenmp-link	This option controls whether the compiler links to static or dynamic OpenMP run-time libraries. To link to the static OpenMP run-time library (RTL) and create a purely static executable, you must specify -openmp-link static (Linux and Mac OS X) or /Qopenmp-link (Windows). However, we strongly recommend you use the default setting, -openmp-link dynamic (Linux and Mac OS X) or /Qopenmp-link:dynamic (Windows).
-openmp-threadprivate	/Qopenmp-threadprivate	This option lets you specify an OpenMP* threadprivate implementation. The legacy OpenMP run-time library is not compatible with object files created using OpenMP run-time libraries supported in other compilers.

When both -openmp and -parallel (Linux OS and Mac OS X) or /Qopenmp and /Qparallel (Windows OS) are specified on the command line, the parallel option is only applied in loop nests that do not have OpenMP directives. For loop nests with OpenMP directives, only the openmp option is applied.

Refer to the following topics for information about OpenMP environment variable and run-time routines:

- OpenMP Environment Variables
- OpenMP Run-time Library Routines
- Intel Extension Routines to OpenMP

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

OpenMP* Source Compatibility and Interoperability with Other Compilers

Intel compilers include two sets of OpenMP libraries:

- The Compatibility OpenMP libraries, which provide compatibility with OpenMP support provided by certain versions of the Microsoft Visual C++* compiler on Windows* OS, certain versions of the GNU* compilers on Linux* OS and Mac OS* X, as well as the Intel compiler version 10.*x* (and later).
- The Legacy OpenMP libraries, which provide compatibility with OpenMP support provided by Intel compilers, including Intel compilers prior to version 10.0.

To select the Compatibility (default) or Legacy OpenMP libraries, use the Intel compiler to link your application and specify the Intel compiler option `/Qopenmp-lib` (Windows OS) or `-openmp-lib` (Linux OS and Mac OS X) .

The term "object-level interoperability" refers to the ability to link object files and libraries generated by one compiler with object files and libraries generated by the second compiler, such that the resulting executable runs successfully. In contrast, "source compatibility" means that the entire application is compiled and linked by one compiler, and you do not need to modify the sources to get the resulting executable to run successfully.

Different compilers support different versions of the OpenMP specification. Based on the OpenMP features your application uses, determine what version of the OpenMP specification your application requires. If your application uses an OpenMP specification level equal or less than the OpenMP specification level supported by all the compilers, your application should have source compatibility with all compilers, but you need to link all object files and libraries with the same compiler's OpenMP libraries.

OpenMP Compatibility Libraries Provided by Intel Compilers

The Compatibility libraries provide source compatibility and object-level interoperability with the OpenMP support provided by:

- On Windows* OS, certain versions of Microsoft Visual C++* that support OpenMP, starting with Microsoft Visual C++ 2005.
- On Linux* OS and Mac OS* X, certain versions of GNU* `gcc`* that support OpenMP, starting with GNU* `gcc`* version 4.2.
- Intel compilers versions 10.0 and later and their supplied OpenMP libraries.

For Fortran applications on Linux systems, it is not possible to link objects compiled by the Intel® Fortran Compiler (`ifort`) with objects compiled by the GNU* Fortran compiler (`gfortran`). Thus, for mixed-language C++ and Fortran applications, you can do one of the following:

- Combine objects created by `gfortran` and Intel® C++ objects, if you specify the Intel OpenMP Compatibility libraries during linking.
- Combine objects created by the Intel C++ compiler and the Intel Fortran Compiler, using Intel OpenMP Compatibility or Legacy libraries.

OpenMP Legacy Libraries Provided by Intel Compilers

The set of Legacy OpenMP libraries has been provided by Intel compilers for multiple releases and provide source compatibility and object-level interoperability with the current Legacy libraries and OpenMP libraries provided by previous Intel compiler versions, including those prior to version 10.0. The Legacy libraries are not compatible with OpenMP support from non-Intel compilers, such as Microsoft Visual C++*, GNU `gcc`*, or GNU Fortran.

You should only use the Legacy libraries if your application requires object-level interoperability with OpenMP library versions provided prior to Intel compilers version 10.0.

Guidelines for Using Different Intel Compiler Versions

To avoid possible linking or run-time problems, follow these guidelines:

- If you compile your application using only the Intel compilers, avoid mixing the Compatibility and Legacy OpenMP runtime libraries. That is, you must link the entire application with either the Compatibility or Legacy libraries.
- When using the Legacy libraries, use the most recent Intel compiler to link the entire application. However, be aware that the Legacy libraries are deprecated, so for a future release, you will need to link the entire application with the Compatibility libraries.
- Use dynamic instead of static OpenMP libraries to avoid linking multiple copies of the libraries into a single program. For details, see OpenMP Support Libraries.

Guidelines for Using Intel and Non-Intel Compilers

To avoid possible linking or run-time problems, follow these guidelines:

- Always link the entire application using the Intel compiler OpenMP Compatibility libraries. This avoids linking multiple copies of the OpenMP runtime libraries from different compilers. It is easiest if you use the Intel compiler command (driver) to link the application, but it is possible to link with the Intel compiler OpenMP Compatibility libraries when linking the application using the GNU* or Visual C++ compiler (or linker) commands.
- If possible, compile all the OpenMP sources with the same compiler. If you compile (not link) using multiple compilers such as the Microsoft Visual C++* or GNU compilers that provide object-level interoperability with the Compatibility libraries, see the instructions in Using the OpenMP Compatibility Libraries.
- Use dynamic instead of static OpenMP libraries to avoid linking multiple copies of the libraries into a single program. For details, see OpenMP Support Libraries.

Limitations When Using OpenMP Compatibility Libraries with Other Compilers

Limitations of threadprivate objects on object-level interoperability:

- On Windows OS systems, the Microsoft Visual C++* compiler uses a different mechanism than the Intel compilers to reference threadprivate data. If you declare a variable as threadprivate in your code and you compile the code with both Intel compilers and Visual C++ compilers, the code compiled by the Intel compiler and the code compiled by the Visual C++* compiler will reference different locations for the variable even when referenced by the same thread. Thus, use the same compiler to compile all source modules that use the same threadprivate objects.
- On Linux OS systems, the GNU* compilers use a different mechanism than the Intel compilers to reference threadprivate data. If you declare a variable as threadprivate in your code and you compile the code with both Intel compilers and GNU compilers, the code compiled by the Intel compiler and the code compiled by the GNU compiler will reference different locations for the variable even when referenced by the same thread. Thus, use the same compiler to compile all source modules that use the same threadprivate objects.
- On Mac OS* X systems, the operating system does not currently support the mechanism used by the GNU* compiler to support threadprivate data. Threadprivate data objects will only be accessible by name from object files compiled by the Intel compilers.

Using OpenMP*

Using OpenMP* in your application requires several steps. To use OpenMP, you must do the following:

1. Add OpenMP directives to your application source code.
2. Compile the application with `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option.
3. For applications with large local or temporary arrays, you may need to increase the stack space available at run-time. In addition, you may need to increase the stack allocated to individual threads by using the KMP_STACKSIZE environment variable or by setting the corresponding library routines.

You can set other environment variables for the multi-threaded code execution.

Add OpenMP Support to the Application

Add the OpenMP API routine declarations to your application by adding a statement similar to the following in your code:

Example
`use omp_lib`

OpenMP Directive Syntax

OpenMP directives use a specific format and syntax. Intel Extension Routines to OpenMP* describes the OpenMP extensions to the specification that have been added to the Intel® compiler.

The following syntax illustrates using the directives in your source.

Example
`<prefix> <directive> [<clause>[[,]<clause>...]]`

where:

- <*prefix*> - Required for all OpenMP directives. For free form source input, the prefix is `!$OMP` only; for fixed form source input, the prefix is `!$OMP` or *`C$OMP`*.
- <*directive*> - A valid OpenMP directive. Must immediately follow the prefix; for example: `!$OMP PARALLEL`.
- [<*clause*>] - Optional. Clauses can be in any order and repeated as necessary, unless otherwise restricted.
- [<*newline*>] - A required component of directive syntax. It precedes the structured block which is enclosed by this directive.
- [,]: Optional. Commas between more than one <*clause*> are optional.

The directives are interpreted as comments if you omit the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows*) option.

The OpenMP constructs defining a parallel region have one of the following syntax forms:

Example

`!$OMP <directive>`
`  <structured block of code>`
`!$OMP END <directive>`
`or`
`!$OMP <directive>`
`  <structured block of code>`
`or`
`!$OMP <directive>`

The following example demonstrates one way of using an OpenMP directive to parallelize a loop.

Example

```
subroutine simple_omp(a, N)
  use omp_lib
  integer :: N, a(N)
!$OMP PARALLEL DO
  do i = 1, N
    a(i) = i*2
  end do
end subroutine simple_omp
```

See OpenMP* Examples for more examples on using directives in specific circumstances.

Compile the Application

The `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option enables the parallelizer to generate multi-threaded code based on the OpenMP directives in the source. The code can be executed in parallel on single processor, multi-processor, or multi-core processor systems.

NOTE. IA-64 Architecture: Specifying this option implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

The `openmp` option works with both `-O0` (Linux and Mac OS X) and `/Od` (Windows) and with any optimization level of `-O1`, `-O2` and `-O3` (Linux and Mac OS X) or `/O1`, `/O2` and `/O3` (Windows).

Specifying `-O0` (Linux and Mac OS X) or `/Od` (Windows) with the OpenMP option helps to debug OpenMP applications.

Compile your application using commands similar to those shown below:

Operating System	Description
Linux and Mac OS X	`ifort -openmp source_file`
Windows	`ifort /Qopenmp source_file`

Assume that you compile the sample above, using the commands similar to the following, where `-c` (Linux and Mac OS X) or `/c` (Windows) instructs the compiler to compile the code without generating an executable:

Operating System	Example
Linux and Mac OS X	`ifort -openmp -c parallel.f90`
Windows	ifort /Qopenmp /c parallel.f90

The compiler might return a message similar to the following:

```
parallel.f90(20) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

Configure the OpenMP Environment

Before you run the multi-threaded code, you can set the number of desired threads using the OpenMP environment variable, `OMP_NUM_THREADS`. See the OpenMP Environment Variables.

Parallel Processing Model

A program containing OpenMP* API compiler directives begins execution as a single thread, called the initial thread of execution. The initial thread executes sequentially until the first parallel construct is encountered.

In the OpenMP API, the `PARALLEL` and `END PARALLEL` directives define the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads, with the initial thread becoming the master of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens the team is dissolved, and only the master thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution.

The following example illustrates, from a high level, the execution model for the OpenMP constructs. The comments in the code explain the structure of each construct or section.

Example

```
PROGRAM MAIN           ! Begin serial execution.
  ...                  ! Only the initial thread executes.
 !$OMP PARALLEL        ! Begin a Parallel construct, form a team.
   ...                 ! This code is executed by each team member.
  !$OMP SECTIONS       ! Begin a worksharing construct.
    !$OMP SECTION      ! One unit of work.
     ...               !
    !$OMP SECTION      ! Another unit of work.
     ...               !
   !$OMP END SECTIONS  ! Wait until both units of work complete.
   ...                 ! More Replicated Code.
  !$OMP DO             ! Begin a worksharing construct,
     DO                ! each iteration is a unit of work.
     ...               ! Work is distributed among the team.
     END DO            !
  !$OMP END DO NOWAIT  ! End of worksharing construct, NOWAIT
                       ! is specified (threads need not wait).
                       ! This code is executed by each team member.
  !$OMP CRITICAL       ! Begin critical construct.
     ...               ! One thread executes at a time.
  !$OMP END CRITICAL   ! End the critical construct.
   ...                 ! This code is executed by each team member.
  !$OMP BARRIER        ! Wait for all team members to arrive.
   ...                 ! This code is executed by each team member.
 !$OMP END PARALLEL    ! End of parallel onstruct, disband team.
```

Example

```
                        ! and continue with serial execution.
  ...                   ! Possibly more parallel constructs.
END PROGRAM MAIN        ! End serial execution.
```

Using Orphaned Directives

In routines called from within parallel constructs, you can also use directives. Directives that are not in the static extent of the parallel construct, but are in the dynamic extent, are called orphaned directives. Orphaned directives allow you to execute portions of your program in parallel with only minimal changes to the sequential version of the program. Using this functionality, you can code parallel constructs at the top levels of your program call tree and use directives to control execution in any of the called routines. For example:

Example

```
subroutine F
...
!$OMP PARALLEL...
   call G
...
subroutine G
!$OMP DO... ! This is an orphaned directive.
...
```

This is an orphaned `DO` loop directive since the parallel region is not lexically present in `subroutine G`.

Data Environment Controls

You can control the data environment within parallel and worksharing constructs. Using directives and data environment clauses on directives, you can privatize named global-lifetime objects by using `THREADPRIVATE` directive, or control data scope attributes by using the data environment clauses for directives that support them.

The data scope attribute clauses are:

- DEFAULT
- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- SHARED

The data copying clauses are:

- COPYIN
- COPYPRIVATE

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them; however, if you do not specify a data scope attribute clause on a directive, the behavior for the variable is determined by the default scoping rules, which are described in the OpenMP API specification, for the variables affected by the directive.

Determining How Many Threads to Use

For applications where the workload depends on application input that can vary widely, delay the decision about the number of threads to employ until runtime when the input sizes can be examined. Examples of workload input parameters that affect the thread count include things like matrix size, database size, image/video size and resolution, depth/breadth/bushiness of tree based structures, and size of list based structures. Similarly, for applications designed to run on systems where the processor count can vary widely, defer the number of threads to employ until application run-time when the machine size can be examined.

For applications where the amount of work is unpredictable from the input data, consider using a calibration step to understand the workload and system characteristics to aid in choosing an appropriate number of threads. If the calibration step is expensive, the calibration results can be made persistent by storing the results in a permanent place like the file system.

Avoid simultaneously using more threads than the number of processing units on the system. This situation causes the operating system to multiplex the processors and typically yields sub-optimal performance.

When developing a library as opposed to an entire application, provide a mechanism whereby the user of the library can conveniently select the number of threads used by the library, because it is possible that the user has higher-level parallelism that renders the parallelism in the library unnecessary or even disruptive.

Use the `NUM_THREADS` clause on parallel regions to control the number of threads employed and use the if clause on parallel regions to decide whether to employ multiple threads at all. The `OMP_SET_NUM_THREADS` routine can also be used, but it also affects parallel regions created by the calling thread. The `NUM_THREADS` clause is local in its effect, so it does not impact other parallel regions.

By default, the Intel OpenMP runtime lets you to create a large number of threads and active nested parallel regions. Use `OMP_GET_THREAD_LIMIT()` and `OMP_GET_MAX_ACTIVE_LEVELS()` to determine the limits. Developers should carefully consider their thread usage and nesting of parallelism to avoid overloading the system. The `OMP_THREADS_LIMIT` environment variable limits the number of OpenMP threads to use for the whole OpenMP program. The `OMP_MAX_ACTIVE_LEVELS` environment variable limits the number of active nested parallel regions.

Verifying OpenMP* Using Parallel Lint

To accelerate migration of sequential applications to parallel applications using OpenMP, parallel lint can be very helpful by reducing application development and debugging time. This topic explains how to use parallel lint to optimize your parallel application. Parallel lint performs static

global analysis of a program to diagnose existing and potential issues with parallelization. One of the advantages of parallel lint is that it makes its checks considering the whole stack of parallel regions and worksharing constructs, even when placed in different routines.

Example

```
1        parameter (N = 100)
2        real, dimension(N) :: x,y
3
4   !$OMP PARALLEL DEFAULT(SHARED)
5   !$OMP SECTIONS
6   !$OMP SECTION
7        do i = 1, N
8        call work(x, N, i)
9        call output(x, N)
10        end do
11   !$OMP SECTION
12        call work(y, N, N)
13        call output(y, N)
14   !$OMP END SECTIONS
15   !$OMP END PARALLEL
16       print *, x, y
17       end
18
19
20       subroutine work(x, N, i)
21         real, dimension(N) :: x
22         x(i) = i*10.0
23       end subroutine work
24
```

Example

```
25        subroutine output(x, N)
26        real, dimension(N) :: x
27  !$OMP SINGLE
28          print *, x
29  !$OMP END SINGLE
tst.f(27): error #12200: SINGLE directive is not allowed in
the dynamic extent of SECTIONS directive (file:tst.f line:5)
```

This makes parallel lint a powerful tool for diagnosing OpenMP directives in whole program context. Parallel lint also provides checks to debug errors connected with data dependencies and race conditions.

Example

```
1         parameter (N = 10)
2         integer i
3         integer, dimension(N) :: factorial
4
5         factorial(1) = 1
6   !$OMP PARALLEL DO
7         do i = 2, N
8             factorial(i) = i * factorial(i-1)
9         end do
10         print *, factorial
11         end
tst.f(8): warning #12246: flow data dependence from
(file:tst.f line:8) to (file:tst.f line:8), due to
"FACTORIAL" may lead to incorrect program execution in parallel mode
```

Basics of Compilation

To enable parallel lint analysis, pass the `/Qdiag-enable:sc-parallel[n]` (Windows), `-diag-enable sc-parallel[n]` (Linux and Mac OS) option to the compiler.

Parallel lint is available for IA-32 and Intel® 64 architectures only.

Parallel lint requires the OpenMP option, `/Qopenmp` (Windows) `-openmp` (Linux and Mac OS). This option forces the compiler to process OpenMP directives to make parallelization specifics available for parallel lint analysis. If parallel lint is used without OpenMP, the compiler issues the following error message:

```
command line error: parallel lint not called due to lack of OpenMP

parallelization option, please add option /Qopenmp when using parallel lint.
```

If you are using Microsoft Visual Studio*, you should create a separate build configuration devoted to parallel lint, since object and library files produced by parallel lint should not be used to build your product.

Basic Checks

Parallel lint provides a broad set of OpenMP checks which are useful both for beginners in parallel programming using OpenMP and for advanced parallel developers. See the Overview section of this manual.

The examples below highlight the most useful features of parallel lint.

Case 1: Nested Regions

An OpenMP program is much more difficult to debug if it has nested parallel regions. Various restrictions apply to nested parallel constructs. Parallel lint can check nested parallel statements even if they are located in different files.

In the example below, a worksharing construct may not be closely nested inside a WORKSHARING, CRITICAL, ORDERED, or MASTER construct.

Example

```
1        parameter (N = 10)
2        real, dimension(N,N) :: x, y, z
3        x = 1.0
4        y = 2.0
5  !$OMP PARALLEL DEFAULT(SHARED)
6  !$OMP MASTER
7        call work(x, y, z, N)
8  !$OMP END MASTER
9  !$OMP END PARALLEL
10        print *, z
11        end
12
13        subroutine work(x, y, z, N)
14        real, dimension(N,N) :: x, y, z
15  !$OMP DO
16        do i = 1, N
17          do j = 1, N
18            z(i,j) = x(i,j) + y(j,i)
19          end do
20        end do
21        end subroutine work
tst.f(15): error #12200: LOOP directive is not allowed in
the dynamic extent of MASTER directive (file:tst.f line:6)
```

Case 2: Data-Sharing Attribute Clauses

Parallelization of an existing serial application requires accurate placement of data sharing clauses. Parallel lint can help determine not only improper usage of sharing clauses but also lack of proper data sharing directives.

The example below demonstrates the OpenMP standard restriction: "If the LASTPRIVATE clause is used on a construct to which NOWAIT is also applied, then the original list item remains undefined until a barrier synchronization has been performed to ensure that the thread that executed the sequentially last iteration, or the lexically last SECTION construct, has stored that list item." [OpenMP standard]

Example

```
1        integer, parameter :: N=10
2        integer last, i
3        real, dimension(N) :: a, b, c
4        b = 10.0
5        c = 50.0
6  !$OMP PARALLEL SHARED(a, b, c, last)
7  !$OMP DO LASTPRIVATE(last)
8        do i = 1, N
9          a(i) = b(i) + c(i)
10          last = i
11        end do
12  !$OMP END DO NOWAIT
13  !$OMP SINGLE
14        call sub(last)
15  !$OMP END SINGLE
16  !$OMP END PARALLEL
17        end
18
19        subroutine sub(last)
20        integer last
21        print *, last
22        end subroutine sub
tst.f(14): error #12220: LASTPRIVATE variable "LAST" in NOWAIT
```

Example

```
work-sharing construct is used before barrier synchronization
```

The next example demonstrates OpenMP standard restriction: "Private pointers that become allocated during the execution of parallel region should be explicitly deallocated by the program prior to the end of parallel region to avoid memory leaks."

Example

```
1        integer :: OMP_GET_THREAD_NUM
2        integer, pointer :: ptr
3        integer, pointer :: a(:)
4
5        call OMP_SET_NUM_THREADS(2)
6        allocate(ptr)
7        allocate(a(2))
8        ptr = 5
9        print *, ptr
10  !$OMP PARALLEL PRIVATE(ptr) SHARED(a)
11        allocate(ptr)
12        ptr = 3
13  !$OMP CRITICAL
14        a(OMP_GET_THREAD_NUM()+1) = ptr
15  !$OMP END CRITICAL
16  !$OMP END PARALLEL
17        print *, a
18        end
as_44_1.f(11): error #12359: private pointer "PTR" should be explicitly deallocated by
the
program prior to the end of parallel region (file:as_44_1.f line:10) to avoid memory
leaks.
```

Case 3: Data Dependence

Data dependency issues are very difficult to debug in parallel programs due to non-deterministic behavior. Parallel lint is able to determine data dependency issues in programs without executing them.

To turn on data dependency analysis you should specify severity level 3 parallel lint in diagnostics.

Example

```
1        integer i, a(4)
2  !$OMP PARALLEL DO  SHARED(i) NUM_THREADS(4)
3        do i=1,4
4           a(i) = loc(i)
5        end do
6  !$OMP END PARALLEL DO
7        print *,a
8        end
tst.f(3): warning #12246: flow data dependence from
(file:tst.f line:3) to (file:tst.f line:3), due to "I"
may lead to incorrect program execution in parallel mode
```

Case 4: Treadprivate Variables

Example

```
1        integer a(1000)
2  !$OMP THREADPRIVATE(a)
3        integer i, sum
4
5  !$OMP PARALLEL DO
6        do i=1,1000
7           a(i) = i
8        end do
9  !$OMP END PARALLEL DO
10  !$OMP PARALLEL DO REDUCTION(+:sum)
11        do i=10,1000
12           sum = sum + a(i)
13        end do
14  !$OMP END PARALLEL DO
15        print *,sum
16        end
tst.f(12): error #12344: THREADPRIVATE variable "A"
is used in loops with different initial values. See
loops (file:tst.f line:6) and (file:tst.f line:11).
```

Case 5: Reductions

Reductions are widely used in parallel programming, but there are a lot of hidden and explicit restrictions. Parallel lint helps avoid potential problems connected to reductions. In this case explicit constraint from the OpenMP API, variables that appear in a `REDUCTION` clause must be `SHARED` in the enclosing context, is illustrated.

Example

```
1        integer i, j
2        real a
3
4  !$OMP PARALLEL PRIVATE(a)
5        do i = 1, 10
6        call sub(a,i)
7        end do
8  !$OMP SINGLE
9        print *, a
10  !$OMP END SINGLE
11  !$OMP END PARALLEL
12        end
13
14        subroutine sub(a,i)
15        integer i
16        real a
17  !$OMP DO REDUCTION(+: a)
18          do j = 1, 10
19              a = a + i + j
20          end do
21        end subroutine sub
as_35_1.f(17): error #12208: variable "A" must be SHARED in the enclosing
```

Example
`context since it is specified in a REDUCTION clause at (file:as_35_1.f line:4)`

OpenMP* Clauses”

Data Scope Attribute Clauses Overview

You can use several directive clauses to control the data scope attributes of variables for the duration of the construct in which you specify them. If you do not specify a data scope attribute clause on a directive, the default is `SHARED` for those variables affected by the directive.

Each of the data scope attribute clauses accepts a list, which is a comma-separated list of named variables or named common blocks that are accessible in the scoping unit. When you specify named common blocks, they must appear between slashes (/*name*/).

Not all of the clauses are allowed on all directives, but the directives to which each clause applies are listed in the clause descriptions.

The data scope attribute clauses are:

- `COPYIN`
- `DEFAULT`
- `PRIVATE`
- `FIRSTPRIVATE`
- `LASTPRIVATE`
- `REDUCTION`
- `SHARED`

Specifying Schedule Type and Chunk Size

The `SCHEDULE` clause of the `DO` or `PARALLEL DO` directive specifies a scheduling algorithm that determines how iterations of the `DO` loop are divided among and dispatched to the threads of the team. The `SCHEDULE` clause applies only to the current `DO` or `PARALLEL DO` directive.

Within the `SCHEDULE` clause, you must specify a *`schedule type`* and, optionally, a *`chunk size`*. A *`chunk`* is a contiguous group of iterations dispatched to a thread. Chunk size must be a scalar integer expression.

The following list describes the schedule types and how the chunk size affects scheduling:

Schedule Type	Description
STATIC	The iterations are divided into pieces having a size specified by chunk. The pieces are statically dispatched to threads in the team in a round-robin manner in the order of thread number. When chunk is not specified, the iterations are first divided into contiguous pieces by dividing the number of iterations by the number of threads in the team. Each piece is then dispatched to a thread before loop execution begins.
DYNAMIC	The iterations are divided into pieces having a size specified by chunk. As each thread finishes its currently dispatched piece of the iteration space, the next piece is dynamically dispatched to the thread. When no chunk is specified, the default is 1.
GUIDED	The chunk size is decreased exponentially with each succeeding dispatch. Chunk specifies the minimum number of iterations to dispatch each time. If there are less than chunk number of iterations remaining, the rest are dispatched. When no chunk is specified, the default is 1.
AUTO	When SCHEDULE(AUTO) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
RUNTIME	The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by using the OMP_SCHEDULE environment variable. When you specify RUNTIME, you cannot specify a chunk size.

The following list shows which schedule type is used, in priority order:

1. The schedule type specified in the SCHEDULE clause of the current DO or PARALLEL DO directive.
2. If the schedule type for the current DO or PARALLEL DO directive is RUNTIME, the default value specified in the OMP_SCHEDULE environment variable.
3. The compiler default schedule type of STATIC.

The following list shows which chunk size is used, in priority order:

- The chunk size specified in the SCHEDULE clause of the current DO or PARALLEL DO directive.
- For STATIC schedule type, the loop iteration space is divided evenly (approximately) by the number of threads in the team.
- For RUNTIME schedule type, the value specified in the OMP_SCHEDULE environment variable.
- For DYNAMIC and GUIDED schedule types, the default value 1.

COPYIN Clause

Use the COPYIN clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to copy the data in the master thread variable or common block to the thread private copies of the variables or common block. The copy occurs at the beginning of the parallel region. The COPYIN clause applies only to variables or common blocks that have been declared THREADPRIVATE.

You do not have to specify a whole common block to be copied in; you can specify named variables that appear in the THREADPRIVATE common block.

Example

```
INTEGER, SAVE:: X

!OMP THREADPRIVATE(X)

!OMP PARALLEL COPYIN(X)
```

In the following example, the common blocks BLK1 and FIELDS are specified as thread private, but only one of the variables in common block FIELDS is specified to be copied.

Example

```
COMMON /BLK1/ SCRATCH
COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(/BLK1/,ZFIELD)
```

DEFAULT Clause

Use the DEFAULT clause on the PARALLEL, PARALLEL DO, PARALLEL SECTIONS and TASK directives to specify a default data scope attribute for all variables within the lexical extent of a parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause. You can specify only one DEFAULT clause on a directive. The default data scope attribute can be one of the following:

Attribute	Description
PRIVATE	Makes all named objects in the lexical extent of the parallel or task region private to a thread. The objects include common block variables, but exclude THREADPRIVATE variables.
FIRSTPRIVATE	Makes all named objects in the lexical extent of the parallel or task region private to a thread, and initialized from the original object. The objects include common block variables, but exclude THREADPRIVATE variables.
SHARED	Makes all named objects in the lexical extent of the parallel or task region shared among all the threads in the team.
NONE	Declares that there is no implicit default as to whether variables are PRIVATE, FIRSTPRIVATE, or SHARED. You must

Attribute	Description
	explicitly specify the scope attribute for each variable in the lexical extent of the parallel or task region.

If you do not specify the DEFAULT clause, the default is DEFAULT (SHARED). However, loop control variables are always PRIVATE by default.

You can exempt variables from the default data scope attribute by using other scope attribute clauses on the parallel region as shown in the following example:

Example

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED(R) LASTPRIVATE(I)
```

PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses

PRIVATE

Use the PRIVATE clause on the PARALLEL, DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to declare variables to be private to each thread in the team.

The behavior of variables declared PRIVATE is as follows:

- A new object of the same type and size is declared once for each thread in the team, and the new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent, but inside the dynamic extent, of the construct unless they are passed as actual arguments to called routines.

In the following example, the values of *I* and *J* are undefined on exit from the parallel region:

Example

```
INTEGER I,J
I = 1
J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
I = 3
J = J + 2
!$OMP END PARALLEL
PRINT *, I, J
```

FIRSTPRIVATE

Use the FIRSTPRIVATE clause on the PARALLEL, DO, SECTIONS, SINGLE, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

In addition to the PRIVATE clause functionality, private copies of the variables are initialized from the original object existing before the parallel construct.

LASTPRIVATE

Use the LASTPRIVATE clause on the DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to provide a superset of the PRIVATE clause functionality.

When the LASTPRIVATE clause appears on a DO or PARALLEL DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct.

When the LASTPRIVATE clause appears on a SECTIONS or PARALLEL SECTIONS directive, the thread that executes the lexically last section updates the version of the object it had before the construct.

Sub-objects that are not assigned a value by the last iteration of the DO loop or the lexically last SECTION directive are undefined after the construct.

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. You must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially. As shown in the following example, the value of I at the end of the parallel region is equal to *N*+1, as it would be with sequential execution.

Example

```
!$OMP PARALLEL
  !$OMP DO LASTPRIVATE(I
    DO I=1,N
      A(I) = B(I) + C(I)
    END DO
  !$OMP END PARALLEL
CALL REVERSE(I)
```

REDUCTION Clause

Use the REDUCTION clause on the PARALLEL, DO, SECTIONS, PARALLEL DO, and PARALLEL SECTIONS directives to perform a reduction on the specified variables by using an operator or intrinsic as shown:

Example

REDUCTION (operator or intrinsic: *list*)

where:

- Operator can be one of the following: +, *, –, .AND., .OR., .EQV., or .NEQV..
- Intrinsic can be one of the following: MAX, MIN, IAND, IOR, or IEOR.

The specified variables must be named variables of intrinsic type and must be shared in the enclosing context. Deferred-shape and assumed-size arrays are not allowed. A private copy of each specified variable is created for each thread as if you had used the PRIVATE clause. The private copy is initialized to a value that depends on the operator or intrinsic as shown in the following table. The actual initialization value is consistent with the data type of the reduction variable.

Operators/Intrinsics and Initialization Values for Reduction Variables

Operators/Intrinsic	Initialization Value
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Largest representable number
MIN	Smallest representable number
IAND	All bits on
IOR	0
IEOR	0

At the end of the construct to which the reduction applies, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the specified operator.

Except for subtraction, all of the reduction operators are associative and the compiler can freely reassociate the computation of the final value. The partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the `REDUCTION` construct. However, if you use the `REDUCTION` clause on a construct to which `NOWAIT` is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all of the threads have completed the `REDUCTION` clause.

The REDUCTION clause is intended to be used on a region or worksharing construct in which the reduction variable is used only in reduction statements having one of the following forms:

Form	Description
x = x operator expr	x is a scalar variable of intrinsic type. Arrays are allowed.
x = expr operator x	except for subtraction
x = intrinsic (x, expr)	intrinsic is either MAX, MIN, IAND, IOR, or IEOR
x = intrinsic (expr, x)	operator is either +, *, –, /, .AND., .OR., .EQV., or .NEQV.

Some reductions can be expressed in other forms. For instance, a MAX reduction might be expressed as follows:

Example
IF (x .LT. expr) x = expr

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator specified in the REDUCTION clause matches the reduction operation.

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a REDUCTION clause for that directive as shown in the following example:

Example
!$OMP DO REDUCTION(+: A, Y),REDUCTION(.OR.: AM)

The following example shows how to use the REDUCTION clause:

Example

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(A,B),REDUCTION(+: A,B)
  DO I=1,N
  CALL WORK(ALOCAL,BLOCAL)
  A = A + ALOCAL
  B = B + BLOCAL
  END DO
!$OMP END PARALLEL DO
```

SHARED Clause

Use the SHARED clause on the PARALLEL, PARALLEL DO, and PARALLEL SECTIONS directives to make variables shared among all the threads in a team.

In the following example, the variables *X* and *NPOINTS* are shared among all the threads in the team:

Example

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(X,NPOINTS)
  IAM = OMP_GET_THREAD_NUM()
  NP = OMP_GET_NUM_THREADS()
  IPOINTS = NPOINTS/NP
  CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

OpenMP* Directives

Programming with OpenMP*

The Intel® compiler accepts a Fortran program containing OpenMP* directives as input and produces a multithreaded version of the code. When the parallel program begins execution, a single thread exists. This thread is called the initial thread.

The initial thread can become the master thread, of a team, when it encounters a parallel region. The initial thread will continue to process serially until it encounters a parallel region.

Parallel Region

A parallel region is a block of code that must be executed by a team of threads in parallel. In the OpenMP API, a parallel construct is defined by placing OpenMP directive `PARALLEL` at the beginning and directive `END PARALLEL` at the end of the code segment. Code segments thus bounded can be executed in parallel.

A structured block of code is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom.

The compiler supports worksharing and synchronization constructs. Each of these constructs consists of one or two specific OpenMP directives and sometimes the enclosed or following structured block of code.

At the end of the parallel region, threads wait until all team members have arrived. The team is logically disbanded (but may be reused in the next parallel region), and the master thread continues serial execution until it encounters the next parallel region.

Worksharing Construct

A worksharing construct divides the execution of the enclosed code region among the members of the team created on entering the enclosing parallel region. When the master thread enters a parallel region, a team of threads is formed. Starting from the beginning of the parallel region, code is executed by all team members until a worksharing construct is encountered. A worksharing construct divides the execution of the enclosed code among the members of the team that encounter it.

The OpenMP `SECTIONS` or `DO` constructs are defined as worksharing constructs because they distribute the enclosed work among the threads of the current team. A worksharing construct is only distributed if it is encountered during dynamic execution of a parallel region. If the

worksharing construct occurs lexically inside of the parallel region, then it is always executed by distributing the work among the team members. If the worksharing construct is not lexically (explicitly) enclosed by a parallel region (that is, it is orphaned), then the worksharing construct will be distributed among the team members of the closest dynamically-enclosing parallel region, if one exists. Otherwise, it will be executed serially.

When a thread reaches the end of a worksharing construct, it may wait until all team members within that construct have completed their work. When all of the work defined by the worksharing construct is finished, the team exits the worksharing construct and continues executing the code that follows.

A combined parallel/worksharing construct denotes a parallel region that contains only one worksharing construct.

Parallel Processing Directive Groups

The parallel processing directives include the following groups:

Parallel Region Directives

- `PARALLEL` and `END PARALLEL`

Worksharing Construct Directives

- The `DO` and `END DO` directives specify parallel execution of loop iterations.
- The `SECTIONS` and `END SECTIONS` directives specify parallel execution for arbitrary blocks of sequential code. Each `SECTION` is executed once by a thread in the team.
- The `SINGLE` and `END SINGLE` directives define a section of code where exactly one thread is allowed to execute the code; threads not chosen to execute this section ignore the code.

Combined Parallel/Worksharing Construct Directives

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- `PARALLEL DO` and `END PARALLEL DO`
- `PARALLEL SECTIONS` and `END PARALLEL SECTIONS`
- `WORKSHARE` and `PARALLEL WORKSHARE`

Synchronization and MASTER Directives

Synchronization is the interthread communication that ensures the consistency of shared data and coordinates parallel execution among threads. Shared data is consistent within a team of threads when all threads obtain the identical value when the data is accessed. A synchronization construct is used to insure this consistency of the shared data.

The OpenMP synchronization directives are CRITICAL, ORDERED, ATOMIC, FLUSH, and BARRIER.

Directive	Usage
CRITICAL	Within a parallel region or a worksharing construct only one thread at a time is allowed to execute the code within a CRITICAL construct.
ORDERED	Used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code.
ATOMIC	Used to update a memory location in an uninterruptable fashion.
FLUSH	Used to insure that all threads in a team have a consistent view of memory.
BARRIER	Forces all team members to gather at a particular point in code. Each team member that executes a BARRIER waits at the BARRIER until all of the team members have arrived. A BARRIER cannot be used within worksharing or other synchronization constructs due to the potential for deadlock.
MASTER	The directive is used to force execution by the master thread.

See the list of OpenMP Directives and Clauses.

Data Sharing

Data sharing is specified at the start of a parallel region or worksharing construct by using the `SHARED` and `PRIVATE` clauses. All variables in the `SHARED` clause are shared among the members of a team. The application must do the following:

- Synchronize access to these variables.
- Insure that all variables in the `PRIVATE` clause are private to each team member. For the entire parallel region, assuming `t` team members, there are `t+1` copies of all the variables in the PRIVATE clause: one global copy that is active outside parallel regions and a `PRIVATE` copy for each team member.
- Initialize `PRIVATE` variables at the start of a parallel region, unless the `FIRSTPRIVATE` clause is specified. In this case, the `PRIVATE` copy is initialized from the global copy at the start of the construct at which the `FIRSTPRIVATE` clause is specified.
- Update the global copy of a `PRIVATE` variable at the end of a parallel region. However, the `LASTPRIVATE` clause of a `DO` directive enables updating the global copy from the team member that executed serially the last iteration of the loop.

In addition to `SHARED` and `PRIVATE` variables, individual variables and entire common blocks can be privatized using the `THREADPRIVATE` directive.

Orphaned Directives

OpenMP contains a feature called orphaning that dramatically increases the expressiveness of parallel directives. Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit. Directives such as `CRITICAL`, `BARRIER`, `SECTIONS`, `SINGLE`, `MASTER`, `DO`, and `TASK` can occur by themselves in a program unit, dynamically "binding" to the enclosing parallel region at run time.

Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring. Orphaning can also improve performance by enabling a single parallel region to bind with multiple DO directives located within called subroutines. Consider the following code segment:

Example

```
subroutine phase1
  integer :: i
  !$OMP DO PRIVATE(i) SHARED(n)
  do i = 1, 200
    call some_work(i)
  end do
  !$OMP END DO
end
subroutine phase2
  integer :: j
  !$OMP DO PRIVATE(j) SHARED(n)
  do j = 1, 100
    call more_work(j)
  end do
  !$OMP END DO
end
program par
  !$OMP PARALLEL
  call phase1
  call phase2
  !$OMP END PARALLEL
end program par
```

Orphaned Directives Usage Rules

The following orphaned directives usage rules apply:

- Any collective operation (worksharing construct or `BARRIER`) executed inside of a worksharing construct is illegal.
- It is illegal to execute a collective operation (worksharing construct or `BARRIER`) from within a synchronization region (`CRITICAL`/`ORDERED`).
- The opening and closing directives of a directive pair (for example, `DO` and `END DO`) must occur in a single block of the program.
- Private scoping of a variable can be specified at a worksharing construct. Shared scoping must be specified at the parallel region.

Preparing Code for OpenMP Processing

The following are the major stages and steps of preparing your code for using OpenMP. Typically, the first two stages can be done on uniprocessor or multiprocessor systems; later stages are typically done on dual core processor and multiprocessor systems.

Before Inserting OpenMP Directives

Before inserting any OpenMP parallel directives, verify that your code is safe for parallel execution by doing the following:

- Place local variables on the stack. This is the default behavior of the Intel Compiler when `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) is specified.
- Use `-automatic` or `-auto-scalar` (Linux and Mac OS X) or `/automatic` (Windows) to make the locals automatic. This is the default behavior of the Intel® compiler when the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) compiler option is specified. Avoid using the `-save` (Linux and Mac OS X) or `/Qsave` (Windows) option, which inhibits stack allocation of local variables. By default, automatic local variables become shared across threads, so you may need to add synchronization code to ensure proper access by threads.

Analyze

The analysis includes the following major actions:

1. Profile the program to find out where it spends most of its time. This is the part of the program that benefits most from parallelization efforts. This stage can be accomplished using basic PGO options.
2. Wherever the program contains nested loops, choose the outer-most loop, which has very few cross-iteration dependencies.

Restructure

To restructure your program for successful OpenMP implementation, you can perform some or all of the following actions:

- If a chosen loop is able to execute iterations in parallel, introduce a `PARALLEL DO` construct around this loop.
- Try to remove any cross-iteration dependencies by rewriting the algorithm.
- Synchronize the remaining cross-iteration dependencies by placing `CRITICAL` constructs around the uses and assignments to variables involved in the dependencies.
- List the variables that are present in the loop within appropriate `SHARED`, `PRIVATE`, `LASTPRIVATE`, `FIRSTPRIVATE`, or `REDUCTION` clauses.
- List the `DO` index of the parallel loop as `PRIVATE`. This step is optional.
- `COMMON` block elements must not be placed on the `PRIVATE` list if their global scope is to be preserved. The `THREADPRIVATE` directive can be used to privatize to each thread the `COMMON` block containing those variables with global scope. `THREADPRIVATE` creates a copy of the `COMMON` block for each of the threads in the team.
- Any I/O in the parallel region should be synchronized.
- Identify more parallel loops and restructure them.
- If possible, merge adjacent `PARALLEL DO` constructs into a single parallel region containing multiple `DO` directives to reduce execution overhead.

Tune

The tuning process should include minimizing the sequential code in critical sections and load balancing by using the `SCHEDULE` clause or the `OMP_SCHEDULE` environment variable.

NOTE. This step is typically performed on dual core processor and multiprocessor systems.

Combined Parallel and Worksharing Constructs

The combined parallel/worksharing constructs provide an abbreviated way to specify a parallel region that contains a single worksharing construct. The combined parallel/worksharing constructs are:

- `PARALLEL DO` and `END PARALLEL DO`
- `PARALLEL SECTIONS` and `END PARALLEL SECTIONS`
- `PARALLEL WORKSHARE` and `END PARALLEL WORKSHARE`

For more details on these directives, see OpenMP* Fortran Compiler Directives.

PARALLEL DO and END PARALLEL DO

Use the `PARALLEL DO` directive to specify a parallel region that implicitly contains a single `DO` directive. You can specify one or more of the clauses for the `PARALLEL DO` directives.

The following example shows how to parallelize a simple loop. The loop iteration variable is private by default, so it is not necessary to declare it explicitly. The `END PARALLEL DO` directive is optional:

Example

```
subroutine par(a, b, N)
  integer :: i, N, a(N), b(N)
  !$OMP PARALLEL DO
  do i= 1, N
    b(i) = (a(i) + a(i-1)) / 2.0
  end do
  !$OMP END PARALLEL DO
end subroutine par
```

PARALLEL SECTIONS and END PARALLEL SECTIONS

Use the `PARALLEL SECTIONS` directive to specify a parallel region that implicitly contains a single `SECTIONS` directive. You can specify one or more of the clauses for the `PARALLEL SECTIONS` directives.

The last section ends at the `END PARALLEL SECTIONS` directive.

In the following example, subroutines `X_AXIS`, `Y_AXIS`, and `Z_AXIS` can be executed concurrently. The first `SECTION` directive is optional. Note that all `SECTION` directives must appear in the lexical extent of the `PARALLEL SECTIONS`/`END PARALLEL SECTIONS` construct:

Example

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL X_AXIS
!$OMP SECTION
      CALL Y_AXIS
!$OMP SECTION
      CALL Z_AXIS
!$OMP END PARALLEL SECTIONS
```

PARALLEL WORKSHARE and END PARALLEL WORKSHARE

Use the `WORKSHARE` directive to divide work within blocks of worksharing statements or constructs into different units. This directive distributes the work of executing the units to threads of the team so each unit is only executed once.

Use the `PARALLEL WORKSHARE` directive to specify parallel regions, in an abbreviated way, that contain a single `WORKSHARE` directive.

When using either directive, be aware that your code cannot branch in to or out of the block defined by these directives.

Parallel Region Directives

The `PARALLEL` and `END PARALLEL` directives define a parallel region as follows:

Example

```
!$OMP PARALLEL
  ! parallel region
!$OMP END PARALLEL
```

When a thread encounters a parallel region, it creates a team of threads and becomes the master of the team. You can control the number of threads in a team by the use of an environment variable, run-time library call, or using the NUM_THREADS clause, or by combining these means.

Clauses Used

The PARALLEL directive takes an optional comma-separated list of clauses that specify as follows:

- IF: whether the statements in the parallel region are executed in parallel by a team of threads or serially by a single thread.
- NUM_THREADS: number of threads in a team.
- PRIVATE, FIRSTPRIVATE, SHARED, or REDUCTION: variable types
- DEFAULT: variable data scope attribute
- COPYIN: master thread common block values are copied to THREADPRIVATE copies of the common block

Changing the Number of Threads

Once created, the number of threads in the team remains constant for the duration of that parallel region. To explicitly change the number of threads used in the next parallel region, you can call the NUM_THREADS clause, call the OMP_SET_NUM_THREADS run-time library routine from a serial portion of the program, or use the OMP_NUM_THREADS environment variable. The order of precedence, or priority, is NUM_THREADS clause, OMP_SET_NUM_THREADS run-time library routine, then OMP_NUM_THREADS environment variable. For example, calling the NUM_THREADS clause overrides the OMP_SET_NUM_THREADS routine, and so forth.

Assuming you have used the OMP_NUM_THREADS environment variable to set the number of threads to 6, you can change the number of threads between parallel regions as follows:

Example

```
!$OMP PARALLEL
... ! This region is executed by 6 threads.
!$OMP END PARALLEL
CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
... ! This region is executed by 3 threads.
!$OMP PARALLEL
!$OMP PARALLEL NUM_THREADS(4)
... ! This region is executed by 4 threads.
!$OMP END PARALLEL
```

Setting Units of Work

Use the worksharing directives such as DO, SECTIONS, and SINGLE to divide the statements in the parallel region into units of work and to distribute those units so that each unit is executed by one thread.

In the following example, the !$OMP DO and !$OMP END DO directives and all the statements enclosed by them comprise the static extent of the parallel region:

Example

```
!$OMP PARALLEL
  !$OMP DO
    DO I=1,N
      B(I) = (A(I) + A(I-1))/ 2.0
    END DO
  !$OMP END DO
!$OMP END PARALLEL
```

In the following example, the DO and END DO directives and all the statements enclosed by them, including all statements contained in the WORK subroutine, comprise the dynamic extent of the parallel region:

Example

```
!$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO
    DO I=1,N
      CALL WORK(I,N)
    END DO
  !$OMP END DO
!$OMP END PARALLEL
```

Setting Conditional Parallel Region Execution

When an IF clause is present on the PARALLEL directive, the enclosed code region is executed in parallel only if the scalar logical expression evaluates to .TRUE.. Otherwise, the parallel region is serialized. When there is no IF clause, the region is executed in parallel by default.

In the following example, the statements enclosed within the DO and END DO directives are executed in parallel only if there are more than three processors available. Otherwise the statements are executed serially:

Example

```
!$OMP PARALLEL IF (OMP_GET_NUM_PROCS() .GT. 3)
  !$OMP DO
    DO I=1,N
      Y(I) = SQRT(Z(I))
    END DO
  !$OMP END DO
!$OMP END PARALLEL
```

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always executed by a team of one thread.

NOTE. To achieve better performance than sequential execution, a parallel region must contain one or more worksharing constructs so that the team of threads can execute work in parallel. It is the contained worksharing constructs that lead to the performance enhancements offered by parallel processing.

For more details on this directive, see OpenMP* Fortran Compiler Directives.

Synchronization Constructs

Synchronization constructs are used to ensure the consistency of shared data and to coordinate parallel execution among threads.

The synchronization constructs are:

- `ATOMIC` directive
- `BARRIER` directive
- `CRITICAL` directive
- `FLUSH` directive
- `MASTER` directive
- `ORDERED` directive

For more details on these directives, see OpenMP* Fortran Compiler Directives.

ATOMIC Directive

Use the `ATOMIC` directive to ensure that a specific memory location is updated atomically instead of exposing the location to the possibility of multiple, simultaneously writing threads.

This directive applies only to the immediately following statement, which must have one of the following forms:

Form	Description
`x = x operator expr`	*x* is a scalar variable of intrinsic type

Form	Description
`x = expr operator x`	*expr* is a scalar expression that does not reference `x`
`x = intrinsic (x, expr)`	*intrinsic* is either MAX, MIN, IAND, IOR or IEOR
`x = intrinsic (expr, x)`	`operator` is either `+`, `*`, `-`, `/`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`

This directive permits optimization beyond that of a critical section around the assignment. An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section. All of these critical sections must use the same unique name.

Only the load and store of *x* are atomic; the evaluation of `expr` is not atomic. To avoid race conditions, all updates of the location in parallel must be protected by using the `ATOMIC` directive, except those that are known to be free of race conditions. The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator, and assignment.

This restriction applies to the `ATOMIC` directive: All references to storage location *x* must have the same type parameters.

In the following example, the collection of `Y` locations is updated atomically:

Example

```
!$OMP ATOMIC
  Y = Y + B(I)
```

BARRIER Directive

To synchronize all threads within a parallel region, use the `BARRIER` directive. You can use this directive only within a parallel region defined by using the `PARALLEL` directive. You cannot use the `BARRIER` directive within the `DO`, `PARALLEL DO`, `SECTIONS`, `PARALLEL SECTIONS`, and `SINGLE` constructs.

When encountered, each thread waits at the `BARRIER` directive until all threads have reached the directive.

In the following example, the BARRIER directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

Example

```
!$OMP PARALLEL
  !$OMP DO PRIVATE(i)
    DO i = 1, 100
      b(i) = i
    END DO
  !$OMP BARRIER
  !$OMP DO PRIVATE(i)
    DO i = 1, 100
      a(i) = b(101-i)
    END DO
!$OMP END PARALLEL
```

CRITICAL and END CRITICAL

Use the CRITICAL and END CRITICAL directives to restrict access to a block of code, referred to as a critical section, to one thread at a time.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name.

When a thread enters the critical section, a latch variable is set to closed and all other threads are locked out. When the thread exits the critical section at the END CRITICAL directive, the latch variable is set to open, allowing another thread access to the critical section.

If you specify a critical section name in the CRITICAL directive, you must specify the same name in the END CRITICAL directive. If you do not specify a name for the CRITICAL directive, you cannot specify a name for the END CRITICAL directive.

All unnamed CRITICAL directives map to the same name. Critical section names are global to the program.

The following example includes several CRITICAL directives, and illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical section. Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, X_AXIS and Y_AXIS, respectively:

Example

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(X,Y)
  !$OMP CRITICAL(X_AXIS)
   CALL DEQUEUE(IX_NEXT, X)
  !$OMP END CRITICAL(X_AXIS)
    CALL WORK(IX_NEXT, X)
  !$OMP CRITICAL(Y_AXIS)
    CALL DEQUEUE(IY_NEXT,Y)
  !$OMP END CRITICAL(Y_AXIS)
  CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

FLUSH Directive

Use the FLUSH directive to identify a synchronization point at which a consistent view of memory is provided. Thread-visible variables are written back to memory at this point.

To avoid flushing all thread-visible variables at this point, include a list of comma-separated named variables to be flushed. The following example uses the FLUSH directive for point-to-point synchronization between thread 0 and thread 1 for the variable ISYNC:

Example

```
!$OMP PARALLEL DEFAULT(PRIVATE),SHARED(ISYNC)
  IAM = OMP_GET_THREAD_NUM()
  ISYNC(IAM) = 0
  !$OMP BARRIER
  CALL WORK()
  ! Synchronize With My Neighbor
  ISYNC(IAM) = 1
  !$OMP FLUSH(ISYNC)
  ! Wait Till Neighbor Is Done
  DO WHILE (ISYNC(NEIGH) .EQ. 0)
  !$OMP FLUSH(ISYNC)
  END DO
. . .
!$OMP END PARALLEL
```

MASTER and END MASTER

Use the MASTER and END MASTER directives to identify a block of code that is executed only by the master thread.

The other threads of the team skip the code and continue execution. There is no implied barrier at the `END MASTER` directive. In the following example, only the master thread executes the routines `OUTPUT` and `INPUT`:

Example

```
!$OMP PARALLEL DEFAULT(SHARED)
  CALL WORK(X)
  !$OMP MASTER
    CALL OUTPUT(X)
    CALL INPUT(Y)
  !$OMP END MASTER
  CALL WORK(Y)
!$OMP END PARALLEL
```

ORDERED and END ORDERED

Use the `ORDERED` and `END ORDERED` directives within a `DO` construct to allow work within an ordered section to execute sequentially while allowing work outside the section to execute in parallel.

When you use the `ORDERED` directive, you must also specify the `ORDERED` clause on the `DO` directive.

Only one thread at a time is allowed to enter the ordered section, and then only in the order of loop iterations. In the following example, the code prints out the indexes in sequential order:

Example

```
!$OMP DO ORDERED,SCHEDULE(DYNAMIC)
  DO I=LB,UB,ST
    CALL WORK(I)
  END DO
SUBROUTINE WORK(K)
  !$OMP ORDERED
    WRITE(*,*) K
  !$OMP END ORDERED
END
```

THREADPRIVATEthreadprivate Directive

You can make named common blocks private to a thread, but global within the thread, by using the `THREADPRIVATE` directive.

Each thread gets its own copy of the common block with the result that data written to the common block by one thread is not directly visible to other threads. During serial portions and master sections of the program, accesses are to the master thread copy of the common block.

You cannot use a thread private variable in any clause other than the following:

- `COPYIN`
- `COPYPRIVATE`
- `SCHEDULE`
- `NUM_THREADS`
- `IF`

In the following example common blocks BLK1 and FIELDS are specified as thread private:

Example

```
COMMON /BLK1/ SCRATCH
COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/,/FIELDS/)
```

For more details on this directive, see OpenMP* Fortran Compiler Directives.

Worksharing Construct Directives

A worksharing construct must be enclosed dynamically within a parallel region if the worksharing directive is to execute in parallel. No new threads are launched and there is no implied barrier on entry to a worksharing construct.

The worksharing constructs are:

- DO and END DO directives
- SECTIONS, SECTION, and END SECTIONS directives
- SINGLE and END SINGLE directives

For more details on these directives, see OpenMP* Fortran Compiler Directives.

DO and END DO

The DO directive specifies that the iterations of the immediately following DO loop must be dispatched across the team of threads so that each iteration is executed by a single thread. The loop that follows a DO directive cannot be a DO WHILE or a DO loop that does not have loop control. The iterations of the DO loop are dispatched among the existing team of threads.

The DO directive optionally lets you:

- Control data scope attributes
- Use the SCHEDULE clause to specify schedule type and chunk size (see Specifying Schedule Type and Chunk Size)

Clauses Used

The clauses for DO directive specify:

- Whether variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION
- How loop iterations are scheduled onto threads
  - In addition, the ORDERED clause must be specified if the ORDERED directive appears in the dynamic extent of the DO directive.
  - If you do not specify the optional NOWAIT clause on the END DO directive, threads synchronize at the END DO directive. If you specify NOWAIT, threads do not synchronize, and threads that finish early proceed directly to the instructions following the END DO directive.

Usage Rules

- You cannot use a GOTO statement, or any other statement, to transfer control onto or out of the DO construct.
- If you specify the optional END DO directive, it must appear immediately after the end of the DO loop. If you do not specify the END DO directive, an END DO directive is assumed at the end of the DO loop, and threads synchronize at that point.
- The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

SECTIONS, SECTION and END SECTIONS

Use the noniterative worksharing SECTIONS directive to divide the enclosed sections of code among the team. Each section is executed just one time by one thread.

Each section should be preceded with a SECTION directive, except for the first section, in which the SECTION directive is optional. The SECTION directive must appear within the lexical extent of the SECTIONS and END SECTIONS directives.

The last section ends at the END SECTIONS directive. When a thread completes its section and there are no undispatched sections, it waits at the END SECTION directive unless you specify NOWAIT.

The SECTIONS directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

The following example shows how to use the SECTIONS and SECTION directives to execute subroutines X_AXIS, Y_AXIS, and Z_AXIS in parallel. The first SECTIONS directive is optional:

Example

```
!$OMP PARALLEL
  !$OMP SECTIONS
    !$OMP SECTION
      CALL X_AXIS
    !$OMP SECTION
      CALL Y_AXIS
    !$OMP SECTION
      CALL Z_AXIS
  !$OMP END SECTIONS
!$OMP END PARALLEL
```

SINGLE and END SINGLE

Use the SINGLE and END SINGLE directives when you want just one thread of the team to execute the enclosed block of code.

Threads that are not executing the SINGLE construct wait at the END SINGLE directive unless you specify NOWAIT.

The SINGLE directive takes an optional comma-separated list of clauses that specifies which variables are PRIVATE or FIRSTPRIVATE.

When the END SINGLE directive is encountered, an implicit barrier is erected and threads wait until all threads have finished. This can be overridden by using the NOWAIT option.

In the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT:

Example

```
!$OMP PARALLEL DEFAULT(SHARED)
  CALL WORK(X)
  !$OMP BARRIER
  !$OMP SINGLE
    CALL OUTPUT(X)
    CALL INPUT(Y)
  !$OMP END SINGLE
  CALL WORK(Y)
!$OMP END PARALLEL
```

Tasking Directives

The tasking model implemented by the Intel® compiler enables OpenMP* to parallelize a large range of applications. The directives used for tasking are:

- TASK and END TASK
- TASKWAIT

TASK and END TASK

The TASK and END TASK directives define an explicit task region as follows:

Example

```
!$OMP TASK
  ! explicit task region
!$OMP END TASK
```

The binding thread set of the task region is the current parallel team. A task region binds to the innermost enclosing PARALLEL region. When a thread encounters a task construct, a task is generated from the structured block enclosed in the construct. The encountering thread may

immediately execute the task, or defer its execution. A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

Clauses Used

The `TASK` directive takes an optional comma-separated list of clauses. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. These clauses are:

- `IF`: whether the task has to be executed immediately or can be deferred.
- `UNTIED`: whether any thread in the team may resume a suspended task
- `PRIVATE`, `FIRSTPRIVATE`, or `SHARED: variable types`
- `DEFAULT`: variable data scope attribute

This example shows a way to generate N tasks with one thread and execute them with the threads in the parallel team:

Example

```
!$OMP PARALLEL SHARED(DATA)
  !$OMP SINGLE PRIVATE (I)
    DO I = 1, N
      !$OMP TASK FIRSTPRIVATE(I), SHARED(DATA)
        CALL WORK(DATA(I))
      !$OMP END TASK
    END DO
  !$OMP END SINGLE
!$OMP END PARALLEL
```

OpenMP* Advanced Issues

This topic discusses how to use the OpenMP* library functions and environment variables and discusses some guidelines for enhancing performance with OpenMP*.

OpenMP* provides specific function calls, and environment variables. See the following topics to refresh you memory about the primary functions and environment variable used in this topic:

- OpenMP* Run-time Library Routines
- OpenMP* Environment Variables

To use the function calls, include the `omp_lib.h` header file or specify `use omp_lib` to use the module file, which are installed in the `INCLUDE` directory during the compiler installation, and compile the application using the `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) option.

The following example, which demonstrates how to use the OpenMP* functions to print the alphabet, also illustrates several important concepts.

First, when using functions instead of directives, your code must be rewritten; rewrites can mean extra debugging, testing, and maintenance efforts.

Second, it becomes difficult to compile without OpenMP support.

Third, it is very easy to introduce simple bugs, as in the loop (below) that fails to print all the letters of the alphabet when the number of threads is not a multiple of 26.

Fourth, you lose the ability to adjust loop scheduling without creating your own work-queue algorithm, which is a lot of extra effort. You are limited by your own scheduling, which is mostly likely static scheduling as shown in the example.

Example

```
include "omp_lib.h"
integer i
integer LettersPerThread, ThisThreadNum, StartLetter, EndLetter
call omp_set_num_threads(4)
  !$OMP PARALLEL PRIVATE(i)
    ! OMP_NUM_THREADS is not a multiple of 26,
    ! which can be considered a bug in this code.
    LettersPerThread = 26 / omp_get_num_threads()
    ThisThreadNum = omp_get_thread_num()
    StartLetter = 'a'+ThisThreadNum*LettersPerThread
    EndLetter = 'a'+ThisThreadNum*LettersPerThread+LettersPerThread
    DO i = StartLetter, EndLetter - 1
       write( *,FMT='(A)',ADVANCE='NO') char(i)
    END DO
  !$OMP END PARALLEL
write(*,*)
end
```

Debugging threaded applications is a complex process, because debuggers change the run-time performance, which can mask race conditions. Even `print` statements can mask issues, because they use synchronization and operating system functions. OpenMP* itself also adds some complications, because it introduces additional structure by distinguishing private variables and shared variables, and inserts additional code. A specialized debugger that supports OpenMP, such as the Intel® Debugger, can help you to examine variables and step through threaded code. You can also use the Intel® Thread Checker to detect many hard-to-find threading errors analytically. Sometimes, a process of elimination can help identify problems without resorting to sophisticated debugging tools.

Remember that most mistakes are race conditions. Most race conditions are caused by shared variables that really should have been declared private. Start by looking at the variables inside the parallel regions and make sure that the variables are declared private when necessary. Next, check functions called within parallel constructs.

The `DEFAULT(NONE)` clause, shown below, can be used to help find those hard-to-spot variables. If you specify `DEFAULT(NONE)`, then every variable must be declared with a data-sharing attribute clause.

Example

```
!$OMP PARALLEL DO DEFAULT(NONE) PRIVATE(x,y) SHARED(a,b)
```

Another common mistake is using uninitialized variables. Remember that private variables do not have initial values upon entering a parallel construct. Use the `FIRSTPRIVATE` and `LASTPRIVATE` clauses to initialize them only when necessary, because doing so adds extra overhead.

If you still can't find the bug, then consider the possibility of reducing the scope. Try a binary-hunt. Another method is to force large chunks of a parallel region to be critical sections. Pick a region of the code that you think contains the bug and place it within a critical section. Try to find the section of code that suddenly works when it is within a critical section and fails when it is not. Now look at the variables, and see if the bug is apparent. If that still doesn't work, try setting the entire program to run in serial by setting the compiler-specific environment variable `KMP_LIBRARY=serial`.

If the code is still not working, compile it without the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) option to make sure the serial version works.

Performance

OpenMP threaded application performance is largely dependent upon the following things:

- The underlying performance of the single-threaded code.
- CPU utilization, idle threads, and load balancing.
- The percentage of the application that is executed in parallel by multiple threads.
- The amount of synchronization and communication among the threads.
- The overhead needed to create, manage, destroy, and synchronize the threads, made worse by the number of single-to-parallel or parallel-to-single transitions called fork-join transitions.
- Performance limitations of shared resources such as memory, bus bandwidth, and CPU execution units.
- Memory conflicts caused by shared memory or falsely shared memory.

Performance always begins with a properly constructed parallel algorithm or application. For example, parallelizing a bubble-sort, even one written in hand-optimized assembly language, is not a good place to start. Keep scalability in mind; creating a program that runs well on two CPUs is not as efficient as creating one that runs well on *n* CPUs. With OpenMP, the number of threads is chosen by the compiler, so programs that work well regardless of the number of threads are highly desirable. Producer/consumer architectures are rarely efficient, because they are made specifically for two threads.

Once the algorithm is in place, make sure that the code runs efficiently on the targeted Intel® architecture; a single-threaded version can be a big help. Turn off the `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows) option to generate a single-threaded version, or build with `-openmp-stubs` (Linux and Mac OS X) or `/Qopenmp-stubs` (Windows), and run the single-threaded version through the usual set of optimizations.

Once you have gotten the single-threaded performance, it is time to generate the multi-threaded version and start doing some analysis.

Optimizations are really a combination of patience, experimentation, and practice. Make little test programs that mimic the way your application uses the computer resources to get a feel for what things are faster than others. Be sure to try the different scheduling clauses for the parallel sections of code.

OpenMP* Examples

The following examples show how to use several OpenMP* features.

A Simple Difference Operator

This example shows a simple parallel loop where the amount of work in each iteration is different. Dynamic scheduling is used to improve load balancing.

The END DO has a NOWAIT because there is an implicit barrier at the end of the parallel region.

Example

```
subroutine do_1(a,b,n)
  real a(n,n), b(n,n)
  !$OMP PARALLEL SHARED(A,B,N)
     !$OMP DO SCHEDULE(DYNAMIC,1) PRIVATE(I,J)
       do i = 2, n
          do j = 1, i
            b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
          end do
      end do
  !$OMP END DO NOWAIT
!$OMP END PARALLEL
end
```

Two Difference Operators: DO Loop Version

The example uses two parallel loops fused to reduce fork/join overhead. The first `END DO` directive has a `NOWAIT` clause because all the data used in the second loop is different than all the data used in the first loop.

Example

```
subroutine do_2(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP DO SCHEDULE(DYNAMIC,1)
      do i = 2, n
        do j = 1, i
          b(j,i) = ( a(j,i) + a(j,i-1) ) / 2.0
        end do
      end do
    !$OMP END DO NOWAIT
    !$OMP DO SCHEDULE(DYNAMIC,1)
      do i = 2, m
        do j = 1, i
          d(j,i) = ( c(j,i) + c(j,i-1) ) / 2.0
        end do
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

Two Difference Operators: SECTIONS Version

The example demonstrates the use of the SECTIONS directive. The logic is identical to the preceding DO example, but uses SECTIONS instead of DO. Here the speedup is limited to 2 because there are only two units of work whereas in the example above there are n-1 + m-1 units of work.

Example

```
subroutine sections_1(a,b,c,d,m,n)
  real a(n,n), b(n,n), c(m,m), d(m,m)
  !$OMP PARALLEL SHARED(A,B,C,D,M,N) PRIVATE(I,J)
    !$OMP SECTIONS
       !$OMP SECTION
          do i = 2, n
             do j = 1, i
                b(j,i)=( a(j,i) + a(j,i-1) ) / 2.0
             end do
          end do
       !$OMP SECTION
          do i = 2, m
             do j = 1, i
                d(j,i)=( c(j,i) + c(j,i-1) ) / 2.0
             end do
          end do
    !$OMP END SECTIONS NOWAIT
  !$OMP END PARALLEL
end
```

Updating a Shared Scalar

This example demonstrates how to use a `SINGLE` construct to update an element of the shared array `a`. The optional nowait after the first loop is omitted because it is necessary to wait at the end of the loop before proceeding into the `SINGLE` construct.

Example

```
subroutine sp_1a(a,b,n)
  real a(n), b(n)
  !$OMP PARALLEL SHARED(A,B,N) PRIVATE(I)
    !$OMP DO
       do i = 1, n
         a(i) = 1.0 / a(i)
       end do
    !$OMP SINGLE
      a(1) = min( a(1), 1.0 )
    !$OMP END SINGLE
    !$OMP DO
      do i = 1, n
        b(i) = b(i) / a(i)
      end do
    !$OMP END DO NOWAIT
  !$OMP END PARALLEL
end
```

Libraries, Directives, Clauses, and Environmental Variables

OpenMP* Environment Variables

The Intel® Compiler supports OpenMP* environment variables (with the OMP_ prefix) and extensions in the form of Intel-specific environment variables (with the KMP_ prefix).

OpenMP Environment Variables

The syntax examples assume bash on Linux* and Mac OS* X. Use the `set` command for Windows*.

Variable Name	Default	Description and Syntax
`OMP_NUM_THREADS`	Number of processors visible to the operating system.	Sets the maximum number of threads to use for OpenMP* parallel regions if no other value is specified in the application. This environment variable applies to both `-openmp` and `-parallel` (Linux and Mac OS X) or `/Qopenmp` and `/Qparallel` (Windows). Example syntax: `export OMP_NUM_THREADS=value`
`OMP_SCHEDULE`	`STATIC`, no chunk size specified	Sets the run-time schedule type and an optional chunk size. Example syntax: `export OMP_SCHEDULE="kind[,chunk_size]"`
`OMP_DYNAMIC`	`.FALSE.` 0	Enables (`.TRUE.`) or disables (`.FALSE.`) the dynamic adjustment of the number of threads. Example syntax: `export OMP_DYNAMIC=value`

Variable Name	Default	Description and Syntax
OMP_NESTED	.FALSE. 0	Enables (.TRUE.) or disables (.FALSE.)nested parallelism. Example syntax: export OMP_NESTED=value
OMP_STACKSIZE	IA-32 architecture: 2M Intel® 64 and IA-32 Architecures: 4M	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread. Recommended size is 16M. Use the optional suffixes: B (bytes), K (Kilobytes), M (Megabytes), G (Gigabytes), or T (Terabytes) to specify the units. If only value is specified and B, K, M, G, or T is not specified, then size is assumed to be K (Kilobytes). This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP program or parallel programs created using -parallel (Linux and Mac OS X) or /Qparallel (Windows). kmp_{set,get}_stacksize_s() routines set/retrieve the value. kmp_set_stacksize_s() routine must be called from sequential part, before first

Variable Name	Default	Description and Syntax
		parallel region is created. Otherwise, calling `kmp_set_stacksize_s()` has no effect. Related env variables: `KMP_STACKSIZE`. `KMP_STACKSIZE` overrides `OMP_STACKSIZE`. Example syntax: `export OMP_STACKSIZE=value`
`OMP_MAX_ACTIVE_LEVELS`	No enforced limit	Limits the number of simultaneously executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: `-openmp` or `-openmp-profile` or `-parallel`

Variable Name	Default	Description and Syntax
OMP_THREAD_LIMIT	No enforced limit	(Linux and Mac OS X) and /Qopenmp or /Qopenmp-profile or /Qparallel (Windows). omp_get_thread_limit() routine returns the value of the limit. Related env variable: KMP_ALL_THREADS. OMP_THREAD_LIMIT overrides KMP_ALL_THREADS. Example syntax: export OMP_THREAD_LIMIT=value Limits the number of simultaneously executing threads in an OpenMP* program. If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue.

Variable Name	Default	Description and Syntax
		This environment variable is only used for programs compiled with the following options: -openmp or `-openmp-profile` or `-parallel` (Linux and Mac OS X) and `/Qopenmp` or `/Qopenmp-profile` or `/Qparallel` (Windows). omp_get_thread_limit() routine returns the value of the limit. Related environment variable: KMP_ALL_THREADS. Its value overrides OMP_THREAD_LIMIT. Example syntax: `export OMP_THREAD_LIMIT=value`

Intel Environment Variables Extensions

Variable Name	Default	Description
`KMP_ALL_THREADS`	No enforced limit	Limits the number of simultaneously executing threads in an OpenMP* program. If this limit is reached and another native operating system thread encounters OpenMP* API calls or constructs, the program can

Variable Name	Default	Description
		abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue. This environment variable is only used for programs compiled with the following options: `-openmp` or `-openmp-profile` (Linux and Mac OS X) and `/Qopenmp` or `/Qopenmp-profile` (Windows).
`KMP_BLOCKTIME`	200 milliseconds	Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. Use the optional character suffixes: `s` (seconds), `m` (minutes), `h` (hours), or `d` (days) to specify the units. Specify `infinite` for an unlimited wait time. See also the throughput execution mode and the `KMP_LIBRARY` environment variable.

Variable Name	Default	Description
KMP_LIBRARY	throughput	Selects the OpenMP run-time library execution mode. The options for the variable value are throughput, turnaround, and serial.
KMP_SETTINGS	0	Enables (1) or disables (0) the printing OpenMP run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP run-time library.
KMP_STACKSIZE	IA-32 architecture: 2m Intel® 64 and IA-64 architectures: 4m	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Recommended size is 16m. Use the optional suffixes: b (bytes), k (kilobytes), m (megabytes), g (gigabytes), or t (terabytes) to specify the units. This variable does not affect the native operating system threads created by the user program nor the thread executing the sequential part of an OpenMP* program or parallel programs created

Variable Name	Default	Description
		using `-parallel` (Linux and Mac OS X) or `/Qparallel` (Windows).
`KMP_MONITOR_STACKSIZE`	max (32k, system minimum thread stack size)	Sets the number of bytes to allocate for the monitor thread, which is used for book-keeping during program execution. Use the optional suffixes: `b` (bytes), `k` (kilobytes), `m` (megabytes), `g` (gigabytes), or `t` (terabytes) to specify the units.
`KMP_VERSION`	`.FALSE.` 0	Enables (`.TRUE.`) or disables (`.FALSE.`) the printing of OpenMP run-time library version information during program execution.
`KMP_AFFINITY`	`noverbose, respect, granularity=core`	Enables run-time library to bind threads to physical processing units. See Thread Affinity Interface for more information on the default and the affect this environment variable has on the parallel environment.
`KMP_CPUINFO_FILE`	`none`	Specifies an alternate file name for file containing machine topology description. The file must be in the same format as `/proc/cpuinfo`.

GNU Environment Variables Extensions

These environment variables are GNU extensions. They are recognized by the Intel OpenMP compatibility library.

Variable Name	Default	Description
GOMP_STACKSIZE	See OMP_STACKSIZE description	OMP_STACKSIZE overrides GOMP_STACKSIZE. KMP_STACKSIZE overrides OMP_STACKSIZE and GOMP_STACKSIZE
GOMP_CPU_AFFINITY	TBD	

OpenMP* Directives and Clauses Summary

This is a summary of the OpenMP* directives and clauses supported in the Intel® Compiler. For detailed information about the OpenMP API, see the *OpenMP Application Program Interface* Version 2.5 specification, which is available from the OpenMP web site (http://www.openmp.org/).

OpenMP Directives

Directive	Description
PARALLEL END PARALLEL	Defines a parallel region.
TASK END TASK	Defines a task region.
DO END DO	Identifies an iterative worksharing construct in which the iterations of the associated loop should be divided among threads in a team.
SECTIONS END SECTIONS	Identifies a non-iterative worksharing construct that specifies a set of structured blocks that are to be divided among threads in a team.
SECTION	Indicates that the associated structured block should be executed in parallel as part of the enclosing sections construct.

Directive	Description
SINGLE END SINGLE	Identifies a construct that specifies that the associated structured block is executed by only one thread in the team.
PARALLEL DO END PARALLEL DO	A shortcut for a PARALLEL region that contains a single DO directive.
	 **NOTE.** The OpenMP PARALLEL DO or DO directive must be immediately followed by a DO statement (DO-stmt as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the compiler issues a syntax error.
PARALLEL SECTIONS END PARALLEL SECTIONS	Provides a shortcut form for specifying a parallel region containing a single SECTIONS construct.
MASTER END MASTER	Identifies a construct that specifies a structured block that is executed by only the master thread of the team.
CRITICAL[*name*] END CRITICAL[*name*]	Identifies a construct that restricts execution of the associated structured block to a single thread at a time. Each thread waits at the beginning of the critical construct until no other thread is executing a critical construct with the same name argument.
TASKWAIT	Indicates a wait on the completion of child tasks generated since the beginning of the current task.
BARRIER	Synchronizes all the threads in a team. Each thread waits until all of the other threads in that team have reached this point.
ATOMIC	Ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneously writing threads.

Directive	Description
FLUSH [(*list*)]	Specifies a cross-thread sequence point at which the implementation is required to ensure that all the threads in a team have a consistent view of certain objects in memory. The optional *list* argument consists of a comma-separated list of variables to be flushed.
ORDERED END ORDERED	The enclosed structured block is executed in the order in which iterations would be executed in a sequential loop.
THREADPRIVATE (*list*)	Makes the named COMMON blocks or variables private to a thread. The list argument consists of a comma-separated list of COMMON blocks or variables.

OpenMP Clauses

Clause	Description
PRIVATE (*list*)	Declares variables in *list* to be PRIVATE to each thread in a team.
FIRSTPRIVATE (*list*)	Same as PRIVATE, but the copy of each variable in the *list* is initialized using the value of the original variable existing before the construct.
LASTPRIVATE (*list*)	Same as PRIVATE, but the original variables in *list* are updated using the values assigned to the corresponding PRIVATE variables in the last iteration in the DO construct loop or the last SECTION construct.
COPYPRIVATE (*list*)	Uses private variables in *list* to broadcast values, or pointers to shared objects, from one member of a team to the other members at the end of a single construct.
NOWAIT	Specifies that threads need not wait at the end of worksharing constructs until they have completed execution. The threads may proceed past the end of the worksharing constructs as soon as there is no more work available for them to execute.
SHARED (*list*)	Shares variables in *list* among all the threads in a team.

Clause	Description
DEFAULT (*mode*)	Determines the default data-scope attributes of variables not explicitly specified by another clause. Possible values for *mode* are PRIVATE, SHARED, or NONE.
REDUCTION ({*operator*|*intrinsic*}:*list*)	Performs a reduction on variables that appear in *list* with the operator operator or the intrinsic procedure name intrinsic; operator is one of the following: +, *, .AND., .OR., .EQV., .NEQV.; intrinsic refers to one of the following: MAX, MIN, IAND, IOR, or IEOR.
ORDERED END ORDERED	Used in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive.
IF (*expression*)	The enclosed parallel region is executed in parallel only if the *expression* evaluates to .TRUE., otherwise the parallel region is serialized. The expression must be scalar logical.
NUM_THREADS (*expression*)	Requests the number of threads specified by expression for the parallel region. The expressions must be scalar integers.
SCHEDULE (*type*[,*chunk*])	Specifies how iterations of the DO construct are divided among the threads of the team. Possible values for the *type* argument are STATIC, DYNAMIC, GUIDED, and RUNTIME. The optional *chunk* argument must be a positive scalar integer expression.
COLLAPSE (n)	Specifies how many loops are associated with the OpenMP loop construct for collapsing.
COPYIN (*list*)	Provide a mechanism to specify that the master thread data values be copied to the THREADPRIVATE copy of the common blocks or variables specified in *list* at the beginning of the parallel region.

Clause	Description
UNTIED	Indicates that a resumed task does not have to be executed by same thread executing it before it was suspended.

Directives and Clauses Cross-reference

See Data Scope Attribute Clauses Overview.

Directive	Use these Clauses
PARALLEL END PARALLEL	• IF • NUM_THREADS • DEFAULT • PRIVATE • FIRSTPRIVATE • SHARED • COPYIN • REDUCTION
DO END DO	• SCHEDULE • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • REDUCTION • ORDERED • NOWAIT • COLLAPSE
SECTIONS END SECTIONS	• PRIVATE • FIRSTPRIVATE

Directive	Use these Clauses
	• LASTPRIVATE • REDUCTION • NOWAIT
SECTION	None
SINGLE END SINGLE	• PRIVATE • FIRSTPRIVATE • COPYPRIVATE • NOTWAIT
PARALLEL DO END PARALLEL DO	• IF • NUM_THREADS • SCHEDULE • DEFAULT • PRIVATE • FIRSTPRIVATE • LASTPRIVATE • SHARED • COPYIN • REDUCTION • ORDERED • COLLAPSE
PARALLEL SECTIONS END PARALLEL SECTIONS	• IF • NUM_THREADS • DEFAULT • PRIVATE

Directive	Use these Clauses
	• FIRSTPRIVATE • LASTPRIVATE • SHARED • COPYIN • REDUCTION
All others	None

OpenMP* Library Support

OpenMP* Run-time Library Routines

OpenMP* provides several run-time library routines to help you manage your program in parallel mode. Many of these run-time library routines have corresponding environment variables that can be set as defaults. The run-time library routines let you dynamically change these factors to assist in controlling your program. In all cases, a call to a run-time library routine overrides any corresponding environment variable.

This topic provides a summary of the OpenMP run-time library routines. See OpenMP* Support Overview for additional resources; refer to the OpenMP API Version 3.0 specification for detailed information about using these routines.

The following tables specify the interfaces to these routines. (The names for the routines are in user name space.)

Execution Environment Routines

Use these routines to monitor and influence threads and the parallel environment.

Function	Description
`SUBROUTINE OMP_SET_NUM_THREADS(num_threads)` `INTEGER num_threads`	Sets the number of threads to use for subsequent parallel regions created by the calling thread.
`INTEGER FUNCTION OMP_GET_NUM_THREADS()`	Returns the number of threads that are being used in the current parallel region.

Function	Description
	This function does not return the value inherited by the calling thread from the `OMP_SET_NUM_THREADS()` function.
`INTEGER FUNCTION OMP_GET_MAX_THREADS()`	Returns the number of threads available to subsequent parallel regions created by the calling thread. This function returns the value inherited by the calling thread from the `OMP_SET_NUM_THREADS()` function.
`INTEGER FUNCTION OMP_GET_THREAD_NUM()`	Returns the thread number of the calling thread, within the context of the current parallel region..
`INTEGER FUNCTION OMP_GET_NUM_PROCS()`	Returns the number of processors available to the program.
`LOGICAL FUNCTION OMP_IN_PARALLEL()`	Returns `.TRUE.` if called within the dynamic extent of a parallel region executing in parallel; otherwise returns `.FALSE.`.
`SUBROUTINE` OMP_SET_DYNAMIC(dynamic_threads) LOGICAL dynamic_threads	Enables or disables dynamic adjustment of the number of threads used to execute a parallel region. If *`dynamic_threads`* is `.TRUE.`, dynamic threads are enabled. If *`dynamic_threads`* is `.FALSE.`, dynamic threads are disabled. Dynamic threads are disabled by default.
`LOGICAL FUNCTION OMP_GET_DYNAMIC()`	Returns `.TRUE.` if dynamic thread adjustment is enabled, otherwise returns `.FALSE.`.
`SUBROUTINE OMP_SET_NESTED(nested)` `LOGICAL nested`	Enables or disables nested parallelism. If *`nested`* is `.TRUE.`, nested parallelism is enabled. If *`nested`* is `.FALSE.`, nested parallelism is disabled. Nested parallelism is disabled by default.

Function	Description
`LOGICAL FUNCTION OMP_GET_NESTED()`	Returns `.TRUE.` if nested parallelism is enabled, otherwise returns `.FALSE.`.
SUBROUTINE OMP_SET_SCHEDULE(kind,modifier) INTEGER(KIND=omp_sched_t) kind INTEGER modifier	Determines the schedule of a worksharing loop that is applied when 'runtime' is used as schedule kind.

`SUBROUTINE OMP_GET_SCHEDULE(kind,modifier)`
`INTEGER(KIND=omp_sched_t) kind`
`INTEGER modifier`
	Returns the schedule of a worksharing loop that is applied when the 'runtime' schedule is used.

`INTEGER FUNCTION OMP_GET_THREAD_LIMIT()`	Returns the maximum number of simultaneously executing threads in an OpenMP* program.
`SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS(max_active_levels)` `INTEGER max_active_levels`	Limits the number of nested active parallel regions. The call is ignored if negative max_active_levels specified.
`INTEGER FUNCTION OMP_GET_MAX_ACTIVE_LEVELS()`	Returns the maximum number of nested active parallel regions.
`INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()`	Returns the number of nested, active parallel regions enclosing the task that contains the call.
`INTEGER FUNCTION OMP_GET_LEVEL()`	Returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region.
`INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(level)` `INTEGER level`	Returns the thread number of the ancestor at a given nest level of the current thread.

Function	Description
`INTEGER FUNCTION OMP_GET_TEAM_SIZE(level) INTEGER level`	Returns the size of the thread team to which the ancestor belongs.

Lock Routines

Use these routines to affect OpenMP locks.

Function	Description
`SUBROUTINE OMP_INIT_LOCK(lock) INTEGER (KIND=OMP_LOCK_KIND)::lock`	Initializes the lock associated with *lock* for use in subsequent calls.
`SUBROUTINE OMP_DESTROY_LOCK(lock) INTEGER(KIND=OMP_LOCK_KIND)::lock`	Causes the lock specified by *lock* to become undefined or uninitialized. The lock must be initialized and not locked.
`SUBROUTINE OMP_SET_LOCK(lock) INTEGER(KIND=OMP_LOCK_KIND)::lock`	Forces the executing thread to wait until the lock associated with *lock* is available. The thread is granted ownership of the lock when it becomes available. The lock must be initialized.
`SUBROUTINE OMP_UNSET_LOCK(lock) INTEGER(KIND=OMP_LOCK_KIND)::lock`	Releases the executing thread from ownership of the lock associated with *lock*. The behavior is undefined if the executing thread does not own the lock associated with *lock*.
`LOGICAL OMP_TEST_LOCK(lock) INTEGER(KIND=OMP_LOCK_KIND)::lock`	Attempts to set the lock associated with *lock*. If successful, returns `.TRUE.`, otherwise returns `.FALSE.`. The lock must be initialized.
`SUBROUTINE OMP_INIT_NEST_LOCK(lock) INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Initializes the nested lock associated with *lock* for use in the subsequent calls.
`SUBROUTINE OMP_DESTROY_NEST_LOCK(lock)`	Causes the nested lock associated with *lock* to become undefined or uninitialized. The lock must be initialized and not locked.

Function	Description
`INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	
`SUBROUTINE OMP_SET_NEST_LOCK(lock)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Forces the executing thread to wait until the nested lock associated with `lock` is available. The thread is granted ownership of the nested lock when it becomes available. The lock must be initialized.
`SUBROUTINE OMP_UNSET_NEST_LOCK(lock)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Releases the executing thread from ownership of the nested lock associated with `lock` if the nesting count is zero. Behavior is undefined if the executing thread does not own the nested lock associated with `lock`.
`INTEGER OMP_TEST_NEST_LOCK(lock)` `INTEGER(KIND=OMP_NEST_LOCK_KIND)::lock`	Attempts to set the nested lock specified by `lock`. If successful, returns the nesting count, otherwise returns zero.

Timing Routines

Function	Description
`DOUBLE PRECISION FUNCTION OMP_GET_WTIME()`	Returns a double precision value equal to the elapsed wall clock time (in seconds) relative to an arbitrary reference time. The reference time does not change during program execution.
`DOUBLE PRECISION FUNCTION OMP_GET_WTICK()`	Returns a double precision value equal to the number of seconds between successive clock ticks.

Intel Extension Routines to OpenMP*

The Intel® Compiler implements the following group of routines as an extensions to the OpenMP* run-time library:

- Getting and setting the execution environment
- Getting and setting stack size for parallel threads

- Memory allocation
- Getting and setting thread sleep time for the throughput execution mode

The Intel extension routines described in this section can be used for low-level tuning to verify that the library code and application are functioning as intended. These routines are generally not recognized by other OpenMP-compliant compilers, which may cause the link stage to fail in other compiler. These OpenMP routines require that you use the `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) command-line option to execute.

See OpenMP* Run-time Library Routines for details about including support for these declarations in your source, and see OpenMP* Support Libraries for detailed information about execution environment (mode).

In most cases, environment variables can be used in place of the extension library routines. For example, the stack size of the parallel threads may be set using the `OMP_STACKSIZE` environment variable rather than the `KMP_SET_STACKSIZE_S()` library routine.

NOTE. A run-time call to an Intel extension routine takes precedence over the corresponding environment variable setting.

Execution Environment Routines

Function	Description
`SUBROUTINE KMP_SET_DEFAULTS(STRING) CHARACTER*(*) STRING`	Sets OpenMP environment variables defined as a list of variables separated by "|" in the argument.
`SUBROUTINE KMP_SET_LIBRARY_THROUGHPUT()`	Sets execution mode to throughput, which is the default. Allows the application to determine the runtime environment. Use in multi-user environments.
`SUBROUTINE KMP_SET_LIBRARY_TURNAROUND()`	Sets execution mode to turnaround. Use in dedicated parallel (single user) environments.
`SUBROUTINE KMP_SET_LIBRARY_SERIAL()`	Sets execution mode to serial.
`SUBROUTINE KMP_SET_LIBRARY(LIBNUM) INTEGER (KIND=OMP_INTEGER_KIND) LIBNUM`	Sets execution mode indicated by the value passed to the function. Valid values are:

Function	Description
	• 1 - serial mode • 2 - turnaround mode • 3 - throughput mode Call this routine before the first parallel region is executed.

`FUNCTION KMP_GET_LIBRARY()`
`INTEGER (KIND=OMP_INTEGER_KIND)`
`KMP_GET_LIBRARY`
	Returns a value corresponding to the current execution mode: 1 (serial), 2 (turnaround), or 3 (throughput).

Stack Size

For IA-64 architecture it is recommended to always use `KMP_SET_STACKSIZE_S()` and `KMP_GET_STACKSIZE_S()`. The `_S()` variants must be used if you need to set a stack size ≥ 2**31 bytes (2 gigabytes).

Function	Description

`FUNCTION KMP_GET_STACKSIZE_S()`
`INTEGER(KIND=KMP_SIZE_T_KIND) &`
`KMP_GET_STACKSIZE_S`
	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can be changed with `KMP_SET_STACKSIZE_S()` routine, prior to the first parallel region or via the `KMP_STACKSIZE` environment variable.

`FUNCTION KMP_GET_STACKSIZE()` `INTEGER KMP_GET_STACKSIZE`	Provided for backwards compatibility only. Use `KMP_GET_STACKSIZE_S()` routine for compatibility across different families of Intel processors.
`SUBROUTINE KMP_SET_STACKSIZE_S(`*`size`*`)` `INTEGER (KIND=KMP_SIZE_T_KIND)`*`size`*	Sets to *`size`* the number of bytes that will be allocated for each parallel thread to use as its private stack. This value can also be set via the `KMP_STACKSIZE` environment variable. In order for `KMP_SET_STACKSIZE_S()kmp_set_stacksize_s()`

Function	Description
	to have an effect, it must be called before the beginning of the first (dynamically executed) parallel region in the program.
`SUBROUTINE KMP_SET_STACKSIZE_S(size)` `INTEGER size`	Provided for backward compatibility only. Use `KMP_SET_STACKSIZE_S(size)` for compatibility across different families of Intel processors.

Memory Allocation

The Intel® compiler implements a group of memory allocation routines as an extension to the OpenMP* run-time library to enable threads to allocate memory from a heap local to each thread. These routines are: `KMP_MALLOC()`, `KMP_CALLOC()`, and `KMP_REALLOC()`.

The memory allocated by these routines must also be freed by the `KMP_FREE()` routine. While it is legal for the memory to be allocated by one thread and freed by a different thread, this mode of operation has a slight performance penalty.

Working with the local heap might lead to improved application performance since synchronization is not required.

Function	Description

`FUNCTION KMP_MALLOC(size)`
`INTEGER(KIND=KMP_POINTER_KIND)KMP_MALLOC`
`INTEGER(KIND=KMP_SIZE_T_KIND)size`
	Allocate memory block of *size* bytes from thread-local heap.

`FUNCTION KMP_CALLOC(nelem,elsize)`
`INTEGER(KIND=KMP_POINTER_KIND)KMP_CALLOC`
`INTEGER(KIND=KMP_SIZE_T_KIND)nelem`
`INTEGER(KIND=KMP_SIZE_T_KIND)elsize`
	Allocate array of *nelem* elements of size *elsize* from thread-local heap.

`FUNCTION KMP_REALLOC(ptr, size)`
`INTEGER(KIND=KMP_POINTER_KIND)KMP_REALLOC`
`INTEGER(KIND=KMP_POINTER_KIND) ptr`
`INTEGER(KIND=KMP_SIZE_T_KIND)size`
	Reallocate memory block at address *ptr* and *size* bytes from thread-local heap.

Function	Description
SUBROUTINE KMP_FREE(*ptr*) INTEGER (KIND=KMP_POINTER_KIND)*ptr*	Free memory block at address *ptr* from thread-local heap. Memory must have been previously allocated with KMP_MALLOC(), KMP_CALLOC(), or KMP_REALLOC().

Thread Sleep Time

In the throughput execution mode, threads wait for new parallel work at the ends of parallel regions, and then sleep, after a specified period of time. This time interval can be set by the KMP_BLOCKTIME environment variable or by the KMP_SET_BLOCKTIME() function.

Function	Description
FUNCTION KMP_GET_BLOCKTIME(INTEGER KMP_GET_BLOCKTIME	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping, as set either by the KMP_BLOCKTIME environment variable or by KMP_SET_BLOCKTIME().
FUNCTION KMP_SET_BLOCKTIME(*msec*) INTEGER *msec*	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping. This routine affects the block time setting for the calling thread and any OpenMP team threads formed by the calling thread. The routine does not affect the block time for any other threads.

OpenMP* Support Libraries

The Intel® Compiler provides support libraries for OpenMP*. There are several kinds of libraries:

- Performance: supports parallel OpenMP execution.
- Profile: supports parallel OpenMP execution and allows use of Intel® Thread Profiler.
- Stubs: supports serial execution of OpenMP applications.

Each kind of library is available for both dynamic and static linking.

NOTE. The use of static OpenMP libraries is not recommended, because they might cause multiple libraries to be linked in an application. The condition is not supported and could lead to unpredictable results.

This section describes the compatibility libraries and legacy libraries provided with the Intel compiler, as well as the selection of run-time execution modes.

Compatibility Libraries

To use the Compatibility OpenMP libraries, specify the (default) `/Qopenmp-lib:compat` (Windows OS) or `-openmp-lib compat` (Linux OS and Mac OS X) compiler option during linking.

On Linux and Mac OS X systems, to use dynamically linked libraries during linking, specify `-openmp-link=dynamic` option; to use static linking, specify the `-openmp-link=static` option.

On Windows systems, to use dynamically linked libraries during linking, specify the `/MD` and `/Qopenmp-link:dynamic` options; to use static linking, specify the `/MT` and `/Qopenmp-link:static` options.

To provide run-time support for dynamically linked applications, the supplied DLL (Windows OS) or shared library (Linux OS and Mac OS X) must be available to the application at run time.

Performance Libraries

To use these libraries, specify the `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) compiler option.

Operating System	Dynamic Link	Static Link
Linux	`libiomp5.so`	`libiomp5.a`
Mac OS X	`libiomp5.dylib`	`libiomp5.a`
Windows	`libiomp5md.lib` `libiomp5md.dll`	`libiomp5mt.lib`

Profile Libraries

To use these libraries, specify `-openmp-profile` (Linux* and Mac OS* X) or `/Qopenmp-profile` (Windows*) compiler option. These allow you to use Intel® Thread Profiler to analyze OpenMP applications.

Operating System	Dynamic Link	Static Link
Linux	`libiompprof5.so`	`libiompprof5.a`
Mac OS X	`libiompprof5.dylib`	`libiompprof5.a`
Windows	`libiompprof5md.lib` `libiompprof5md.dll`	`libiompprof5mt.lib`

Stubs Libraries

To use these libraries, specify `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) compiler option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libiompstubs5.so`	`libiompstubs5.a`
Mac OS X	`libiompstubs5.dylib`	`libiompstubs5.a`
Windows	`libiompstubs5md.lib` `libiompstubs5md.dll`	`libiompstubs5mt.lib`

Legacy Libraries

To use the Legacy OpenMP libraries, specify the `/Qopenmp-lib:legacy` (Windows OS) or `-openmp-lib legacy` (Linux OS and Mac OS X) compiler options during linking. Legacy libraries are deprecated.

On Linux and Mac OS X systems, to use dynamically linked libraries during linking, specify the `-openmp-link:dynamic` option; to use static linking, specify the `-openmp-link:static` option.

On Windows systems, to use dynamically linked libraries during linking, specify the `/MD` and `/Qopenmp-link=dynamic` options; to use static linking, specify the `/MT` and `/Qopenmp-link=static` options.

To provide run-time support for dynamically linked applications, the supplied DLL (Windows OS) or shared library (Linux OS and Mac OS X) must be available to the application at run time.

Performance Libraries

To use these libraries, specify `-openmp` (Linux* and Mac OS* X) or `/Qopenmp` (Windows*) compiler option.

Operating System	Dynamic Link	Static Link
Linux	`libguide.so`	`libguide.a`
Mac OS X	`libguide.dylib`	`libguide.a`
Windows	`libguide40.lib` `libguide40.dll`	`libguide.lib`

Profile Libraries

To use these libraries, specify `-openmp-profile` (Linux* and Mac OS* X) or `/Qopenmp-profile` (Windows*) compiler option. These allow you to use Intel® Thread Profiler to analyze OpenMP applications.

Operating System	Dynamic Link	Static Link
Linux	`libguide_stats.so`	`libguide_stats.a`
Mac OS X	`libguide_stats.dylib`	`libguide_stats.a`
Windows	`libguide40_stats.lib` `libguide40_stats.dll`	`libguide_stats.lib`

Stubs Libraries

To use these libraries, specify `-openmp-stubs` (Linux* and Mac OS* X) or `/Qopenmp-stubs` (Windows*) compiler option. These allow you to compile OpenMP applications in serial mode and provide stubs for OpenMP routines and extended Intel-specific routines.

Operating System	Dynamic Link	Static Link
Linux	`libompstub.so`	`libompstub.a`
Mac OS X	`libompstub.dylib`	`libompstub.a`

Operating System	Dynamic Link	Static Link
Windows	libompstub40.lib libompstub40.dll	libompstub.lib

Execution modes

The Intel compiler enables you to run an application under different execution modes specified at run time; the libraries support the turnaround, throughput, and serial modes. Use the KMP_LIBRARY environment variable to select the modes at run time.

Mode	Description
throughput (default)	The throughput mode allows the program to detect its environment conditions (system load) and adjust resource usage to produce efficient execution in a dynamic environment. In a multi-user environment where the load on the parallel machine is not constant or where the job stream is not predictable, it may be better to design and tune for throughput. This minimizes the total time to run multiple jobs simultaneously. In this mode, the worker threads yield to other threads while waiting for more parallel work. After completing the execution of a parallel region, threads wait for new parallel work to become available. After a certain period of time has elapsed, they stop waiting and sleep. Until more parallel work becomes available, sleeping allows processor and resources to be used for other work by non-OpenMP threaded code that may execute between parallel regions, or by other applications. The amount of time to wait before sleeping is set either by the KMP_BLOCKTIME environment variable or by the

Mode	Description
turnaround	`KMP_SET_BLOCKTIME()` function. A small blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions. A larger blocktime value may be more appropriate if threads are to be reserved solely for use for OpenMP execution, but may penalize other concurrently-running OpenMP or threaded applications. The turnaround mode is designed to keep active all processors involved in the parallel computation, which minimizes execution time of a single job. In this mode, the worker threads actively wait for more parallel work, without yielding to other threads. In a dedicated (batch or single user) parallel environment where all processors are exclusively allocated to the program for its entire run, it is most important to effectively use all processors all of the time.  **NOTE.** Avoid over-allocating system resources. The condition can occur if either too many threads have been specified, or if too few processors are available at run time. If system resources are over-allocated, this mode will cause poor performance. The throughput mode should be used instead if this occurs.
serial	The serial mode forces parallel applications to run as a single thread.

Using the OpenMP Compatibility Libraries

This section describes the steps needed to set up and use the OpenMP Compatibility Libraries from the command line. On Windows* systems, you can also build applications compiled with the OpenMP Compatibility libraries in the Microsoft Visual Studio* development environment.

For a summary of the support provided by the Compatibility and Legacy libraries provided with Intel compilers, see OpenMP* Source Compatibility and Interoperability with Other Compilers.

For a list of the options and libraries used by the OpenMP libraries, see OpenMP Support Libraries.

Set up your environment for access to the Intel compiler to ensure that the appropriate OpenMP library is available during linking. On Windows systems, you can either execute the appropriate batch (.bat) file or use the command-line window supplied in the compiler program folder that already has the environment set up. On Linux and Mac OS systems, you can source the appropriate script file (see Using the ifortvars File to Specify Location of Components.

During C/C++ compilation, ensure the version of `omp.h` used when compiling is the version provided by that compiler. For example, on Linux systems when compiling with the GNU C/C++ compiler, use the `omp.h` provided with the GNU C/C++ compiler. Similarly, during Fortran compilation, ensure that the version of `omp_lib.h` or `omp_lib.mod` used when compiling is the version provided by that compiler.

The following table lists the commands used by the various command-line compilers for both C and C++ source files.:

Operating System	C Source Module	C++ Source Module
Linux*	GNU: `gcc` Intel: `icc`	GNU: `g++` Intel: `icpc`
Mac OS* X	GNU: `gcc` Intel: `icc`	GNU: `g++` Intel: `icpc`
Windows*	Visual C++: `cl` Intel: `icl`	Visual C++: `cl` Intel: `icl`

The command for the Intel® Fortran compiler is `ifort` on Linux, Mac OS X, and Windows operating systems.

For information on the OpenMP libraries and options used by the Intel compiler, see OpenMP Support Libraries.

Command-Line Examples, Windows OS

On Windows systems, to use the Compatibility Libraries with Microsoft Visual C++ in the Microsoft Visual Studio* environment, see Using the OpenMP Compatibility Libraries from Visual Studio*.

To compile and link (build) the entire application with one command using the Compatibility libraries, specify the following Intel compiler command:

Type of File	Commands
Fortran source, dynamic link	`ifort /MD /Qopenmp /Qopenmp-lib:compat hello.f90`

By default, the Intel compilers perform a dynamic link of the OpenMP libraries. To perform a static link (not recommended), use the `/MT` option in place of the `/MD` option above and add the option `/Qopenmp-link:static`. The Intel compiler option `/Qopenmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Windows OS systems (default is `/Qopenmp-link:dynamic`).

When using Microsoft Visual C++ compiler, you should link with the Intel OpenMP compatibility library. You need to avoid linking the Microsoft OpenMP run-time library (vcomp) and explicitly pass the name of the Intel OpenMP compatibility library as linker options (following `/link`):

Type of File	Commands
C source, dynamic link	`icl /MD /openmp hello.c /link /nodefaultlib:vcomp libiomp5md.lib`
C++ source, dynamic link	`icl /MD /openmp hello.cpp /link /nodefaultlib:vcomp libiomp5md.lib`

Performing a static link is not recommended, but would require use of the `/MT` option in place of the `/MD` option above.

You can also use both Intel C++ and Visual C++ compilers to compile parts of the application and create object files (object-level interoperability). In this example, the Intel compiler links the entire application:

Type of File	Commands

C source, dynamic link	`icl /MD /openmp hello.cpp /c f1.c f2.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat /c f3.c f4.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat f1.obj f2.obj f3.obj f4.obj /Feapp /link /nodefaultlib:vcomp`

The first command produces two object files compiled by Visual C++ compiler, and the second command produces two more object files compiled by Intel C++ Compiler. The final command links all four object files into an application.

Alternatively, the third line below uses the Visual C++ linker to link the application and specifies the Compatibility library `libiomp5md.lib` at the end of the third command:

Type of File	Commands

C source, dynamic link	`icl /MD /openmp hello.cpp /c f1.c f2.c`
	`icl /MD /Qopenmp /Qopenmp-lib:compat /c f3.c f4.c`
	`link f1.obj f2.obj f3.obj f4.obj /out:app.exe /nodefaultlib:vcomp libiomp5md.lib`

The following example shows the use of interprocedural optimization by the Intel compiler on several files, the Visual C++ compiler compiles several files, and the Visual C++ linker links the object files to create the executable:

Type of File	Commands

C source, dynamic link	`icl /MD /Qopenmp /Qopenmp-lib:compat /O3 /Qipo /Qipo-c f1.c f2.c f3.c`
	`cl /MD /openmp /O2 /c f4.c f5.c`
	`cl /MD /openmp /O2 ipo_out.obj f4.obj f5.obj /Feapp /link /nodefaultlib:vcomp libiomp5md.lib`

The first command uses the Intel® C++ Compiler to produce an optimized multi-file object file named `ipo_out.obj` by default (the `/Fe` option is not required; see Using IPO). The second command uses the Visual C++ compiler to produce two more object files. The third command uses the Visual C++ `cl` command to link all three object files using the Intel compiler OpenMP Compatibility library. Performing a static link (not recommended) requires use of the `/MT` option in place of the `/MD` option in each line above.

Command-Line Examples, Linux OS and Mac OS X

To compile and link (build) the entire application with one command using the Compatibility libraries, specify the following Intel compiler command:

Type of File	Commands
Fortran source	`ifort -openmp -openmp-lib=compat hello.f90`

By default, the Intel compilers perform a dynamic link of the OpenMP libraries. To perform a static link (not recommended), add the option `-openmp-link static`. The Intel compiler option `-openmp-link` controls whether the linker uses static or dynamic OpenMP libraries on Linux OS and Mac OS X systems (default is `-openmp-link dynamic`).

You can also use both Intel C++ `icc/icpc` and GNU `gcc/g++` compilers to compile parts of the application and create object files (object-level interoperability). In this example, the GNU compiler compiles the C file `foo.c` (the `gcc` option `-fopenmp` enables OpenMP support), and the Intel compiler links the application using the OpenMP Compatibility library:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c` `icc -openmp -openmp-lib=compat foo.o`
C++ source	`g++ -fopenmp -c foo.cpp` `icpc -openmp -openmp-lib=compat foo.o`

When using GNU `gcc` or `g++` compiler to link the application with the Intel compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP compatibility library name using the `-l` option, the GNU `pthread` library using the `-l` option, and path to the Intel libraries where the Intel C++ compiler is installed using the `-L` option:

Type of File	Commands
C source	`gcc -fopenmp -c foo.c bar.c` `gcc foo.o bar.o -liomp5 -lpthread -L<icc_dir>/lib`

You can mix object files, but it is easier to use the Intel compiler to link the application so you do not need to specify the `gcc -l` option, `-L` option, and the `-lpthread` option:

Type of File	Commands

C source	`gcc -fopenmp -c foo.c`
	`icc -openmp -c bar.c`
	`icc -openmp -openmp=compat foo.o bar.o`

You can mix OpenMP object files compiled with the GNU gcc compiler, the Intel® C++ Compiler, and the Intel® Fortran Compiler. This example uses use the Intel Fortran Compiler to link all the objects:

Type of File	Commands
Mixed C and Fortran sources	`ifort -openmp -c foo.f` `icc -openmp -c ibar.c`

Type of File	Commands
	gcc -fopenmp -c gbar.c ifort -openmp -openmp-lib=compat foo.o ibar.o gbar.o

When using the Intel Fortran compiler, if the main program does not exist in a Fortran object file that is compiled by the Intel Fortran Compiler `ifort`, specify the `-nofor-main` option on the `ifort` command line during linking.

NOTE. Do not mix objects created by the Intel Fortran Compiler (`ifort`) with the GNU Fortran Compiler (`gfortran`); instead, recompile all Fortran sources with the same Fortran compiler. The GNU Fortran Compiler is only available on Linux operating systems.

Similarly, you can mix object files compiled with the Intel® C++ Compiler, the GNU C/C++ compiler, and the GNU Fortran Compiler (gfortran), if you link with the GNU Fortran Compiler (`gfortran`). When using GNU `gfortran` compiler to link the application with the Intel compiler OpenMP compatibility library, you need to explicitly pass the Intel OpenMP compatibility library name and the Intel `irc` libraries using the `-l` options, the GNU `pthread` library using the `-l` option, and path to the Intel libraries where the Intel C++ compiler is installed using the `-L`. option. You do not need to specify the `-fopenmp` option on the link line:

Type of File	Commands
Mixed C and GNU Fortran sources	gfortran -fopenmp -c foo.f icc -openmp -c ibar.c gcc -fopenmp -c gbar.c gfortran foo.o ibar.o gbar.o -lirc -liomp5 -lpthread -lc -L<icc_dir>/lib

Alternatively, you could use the Intel compiler to link the application, but need to pass multiple `gfortran` libraries using the `-l` options on the link line:

Type of File	Commands
Mixed C and Fortran sources	gfortran -fopenmp -c foo.f icc -openmp -c ibar.c icc -openmp -openmp-lib=compat foo.o bar.o -lgfortranbegin -lgfortran

Using the OpenMP Compatibility Libraries from Visual Studio

When using systems running a Windows OS, you can make certain changes in the Visual C++ Visual Studio 2005 development environment to allow you to use the Intel C++ Compiler and Visual C++ to create applications that use the Intel compiler OpenMP Compatibility libraries.

NOTE. Microsoft Visual C++ must have the symbol `_OPENMP_NOFORCE_MANIFEST` defined or it will include the manifest for the `vcomp90` dlls. While this may not appear to cause a problem on the build system, it will cause a problem when the application is moved to another system that does not have this DLL installed.

Set the project Property Pages to indicate the Intel OpenMP run-time library location:

1. Open the project's property pages in from the main menu: Project > Properties (or right click the Project name and select Properties) .
2. Select Configuration Properties > Linker > General > Additional Library Directories
3. Enter the path to the Intel compiler libraries. For example, for an IA-32 architecture system, enter: `< Intel_compiler_installation_path>\IA32\LIB`

Make the Intel OpenMP dynamic run-time library accessible at run-time, you must specify the corresponding path:

1. Open the project's property pages in from the main menu: Project > Properties (or right click the Project name and select Properties).
2. Select Configuration Properties > Debugging > Environment
3. Enter the path to the Intel compiler libraries. For example, for an IA-32 architecture system, enter:

```
PATH=%PATH%;< Intel_compiler_installation_path>\IA32\Bin
```

Add the Intel OpenMP run-time library name to the linker options and exclude the default Microsoft OpenMP run-time library:

1. Open the project's property pages in from the main menu: Project > Properties (or right click the Project name and select Properties).
2. Select Configuration Properties > Linker > Command Line > Additional Options
3. Enter the OpenMP library name and the Visual C++ linker option `/nodefaultlib`:

Thread Affinity Interface (Linux* and Windows*)

The Intel® runtime library has the ability to bind OpenMP threads to physical processing units. The interface is controlled using the `KMP_AFFINITY` environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

`Thread affinity` restricts execution of certain threads (virtual execution units) to a subset of the physical processing units in a multiprocessor computer. Depending upon the topology of the machine, thread affinity can have a dramatic effect on the execution speed of a program.

Thread affinity is supported on Windows OS systems and versions of Linux OS systems that have kernel support for thread affinity, but is not supported by Mac OS* X. The thread affinity interface is supported only for Intel® processors.

The Intel compiler's OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. There are three types of interfaces you can use to specify this binding, which are collectively referred to as the Intel OpenMP Thread Affinity Interface:

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP threads to the processors based upon their physical location in the machine. This interface is controlled entirely by the KMP_AFFINITY environment variable.
- The mid-level affinity interface uses an environment variable to explicitly specifies which processors (labeled with integer IDs) are bound to OpenMP threads. This interface provides compatibility with the GNU gcc* GOMP_CPU_AFFINITY environment variable, but you can also invoke it by using the KMP_AFFINITY environment variable. The GOMP_CPU_AFFINITY environment variable is supported on Linux systems only, but users on Windows or Linux systems can use the similar functionality provided by the KMP_AFFINITY environment variable.
- The low-level affinity interface uses APIs to enable OpenMP threads to make calls into the OpenMP runtime library to explicitly specify the set of processors on which they are to be run. This interface is similar in nature to `sched_setaffinity` and related functions on Linux* systems or to `SetThreadAffinityMask` and related functions on Windows* systems. In addition, you can specify certain options of the KMP_AFFINITY environment variable to affect the behavior of the low-level API interface. For example, you can set the affinity type KMP_AFFINITY to disabled, which disables the low-level affinity interface, or you could use the KMP_AFFINITY or GOMP_CPU_AFFINITY environment variables to set the initial affinity mask, and then retrieve the mask with the low-level API interface.

The following terms are used in this section

- The total number of processing elements on the machine is referred to as the number of *`OS thread contexts.`*
- Each processing element is referred to as an Operating System processor, or *`OS proc`*.

- Each OS processor has a unique integer identifier associated with it, called an *OS proc ID*.
- The term *package* refers to a single or multi-core processor chip.
- The term *OpenMP Global Thread ID* (GTID) refers to an integer which uniquely identifies all threads known to the Intel OpenMP runtime library. The thread that first initializes the library is given GTID 0. In the normal case where all other threads are created by the library and when there is no nested parallelism, then *n-threads-var* - 1 new threads are created with GTIDs ranging from 1 to *ntheads-var* - 1, and each thread's GTID is equal to the OpenMP thread number returned by function `omp_get_thread_num()`. The high-level and mid-level interfaces rely heavily on this concept. Hence, their usefulness is limited in programs containing nested parallelism. The low-level interface does not make use of the concept of a GTID, and can be used by programs containing arbitrarily many levels of parallelism.

The KMP_AFFINITY Environment Variable

The `KMP_AFFINITY` environment variable uses the following general syntax:

Syntax
KMP_AFFINITY=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

For example, to list a machine topology map, specify `KMP_AFFINITY=verbose,none` to use a *modifier* of `verbose` and a *type* of `none`.

The following table describes the supported specific arguments.

Argument	**Default**	**Description**

modifier	`noverbose`
	`respect`
	`granularity=core`
		Optional. String consisting of keyword and specifier.
		• `granularity=`<*specifier*> takes the following specifiers: `fine`, `thread`, and `core`
		• `norespect`
		• `noverbose`
		• `nowarnings`
		• `proclist={`<*proc-list*>`}`
		• `respect`

Argument	Default	Description
		• verbose • warnings The syntax for <*proc-list*> is explained in mid-level affinity interface.
type	none	Required string. Indicates the thread affinity to use. • compact • disabled • explicit • none • scatter • logical (deprecated; instead use compact, but omit any permute value) • physical (deprecated; instead use scatter, possibly with an offset value) The logical and physical types are deprecated but supported for backward compatibility.
permute	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.

Argument	Default	Description
offset	0	Optional. Positive integer value. Not valid with type values of explicit, none, or disabled.

Affinity Types

Type is the only required argument.

type = none (default)

Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify KMP_AFFINITY=verbose,none to list a machine topology map.

type = compact

Specifying compact assigns the OpenMP thread <*n*>+1 to a free thread context as close as possible to the thread context where the <*n*> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

type = disabled

Specifying disabled completely disables the thread affinity interfaces. This forces the OpenMP run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as kmp_set_affinity and kmp_get_affinity, which have no effect and will return a nonzero error code.

type = explicit

Specifying explicit assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the proclist= modifier, which is required for this affinity type. See Explicitly Specifying OS Proc IDs (GOMP_CPU_AFFINITY).

type = scatter

Specifying scatter distributes the threads as evenly as possible across the entire system. scatter is the opposite of compact; so the leaves of the node are most significant when sorting through the machine topology map.

Deprecated Types: logical and physical

Types `logical` and `physical` are deprecated and may become unsupported in a future release. Both are supported for backward compatibility.

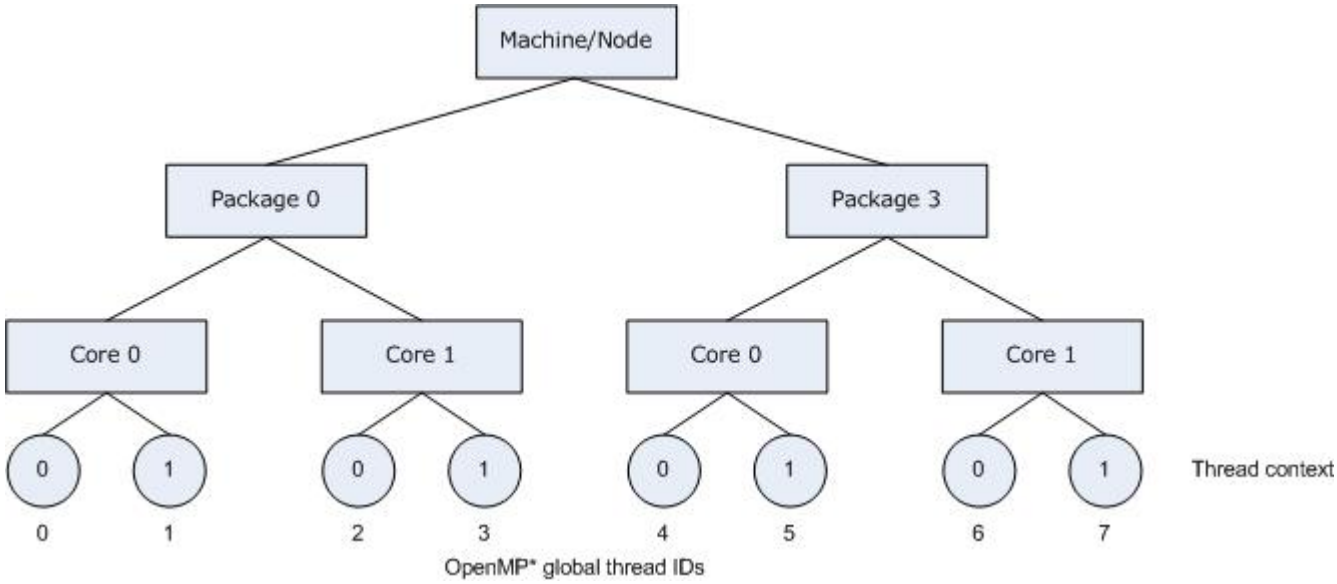
For `logical` and `physical` affinity types, a single trailing integer is interpreted as an `offset` specifier instead of a `permute` specifier. In contrast, with `compact` and `scatter` types, a single trailing integer is interpreted as a `permute` specifier.

- Specifying `logical` assigns OpenMP threads to consecutive logical processors, which are also called hardware thread contexts. The type is equivalent to `compact`, except that the `permute` specifier is not allowed. Thus, `KMP_AFFINITY=logical,n` is equivalent to `KMP_AFFINITY=compact,0,n` (this equivalence is true regardless of the whether or not a `granularity=fine` modifier is present).
- Specifying `physical` assigns threads to consecutive physical processors (cores). For systems where there is only a single thread context per core, the type is equivalent to `logical`. For systems where multiple thread contexts exist per core, `physical` is equivalent to `compact` with a `permute` specifier of 1; that is, `KMP_AFFINITY=physical,n` is equivalent to `KMP_AFFINITY=compact,1,n` (regardless of the whether or not a `granularity=fine` modifier is present). This equivalence means that when the compiler sorts the map it should permute the innermost level of the machine topology map to the outermost, presumably the thread context level. This type does not support the `permute` specifier.

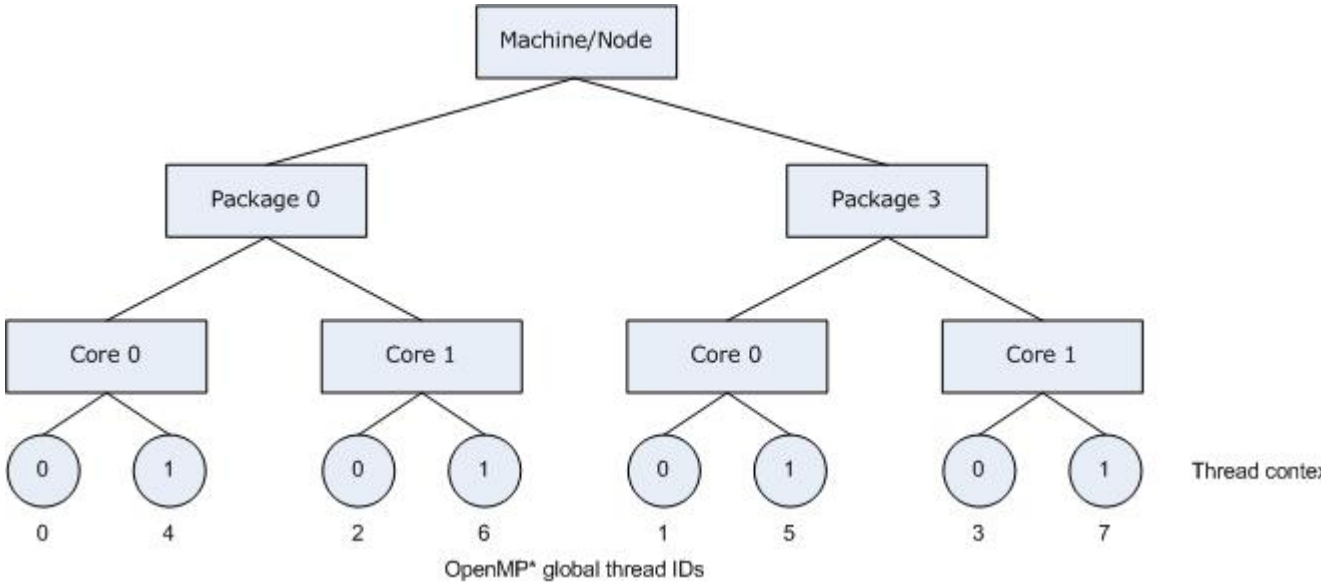
Examples of Types `compact` and `scatter`

The following figure illustrates the topology for a machine with two processors, and each processor has two cores; further, each core has Hyper-Threading Technology (HT Technology) enabled.

The following figure also illustrates the binding of OpenMP thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`.



Specifying `scatter` on the same system as shown in the figure above, the OpenMP threads would be assigned the thread contexts as shown in the following figure, which shows the result of specifying `KMP_AFFINITY=granularity=fine,scatter`.



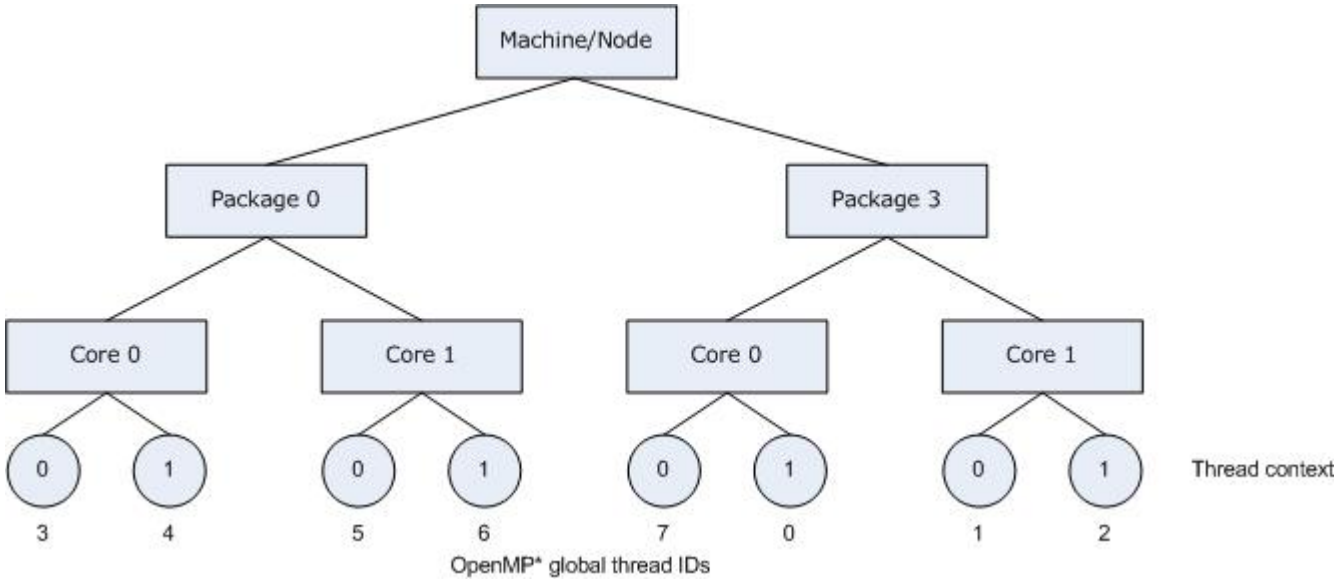
permute and offset combinations

For both `compact` and `scatter`, `permute` and `offset` are allowed; however, if you specify only one integer, the compiler interprets the value as a permute specifier. Both `permute` and `offset` default to 0.

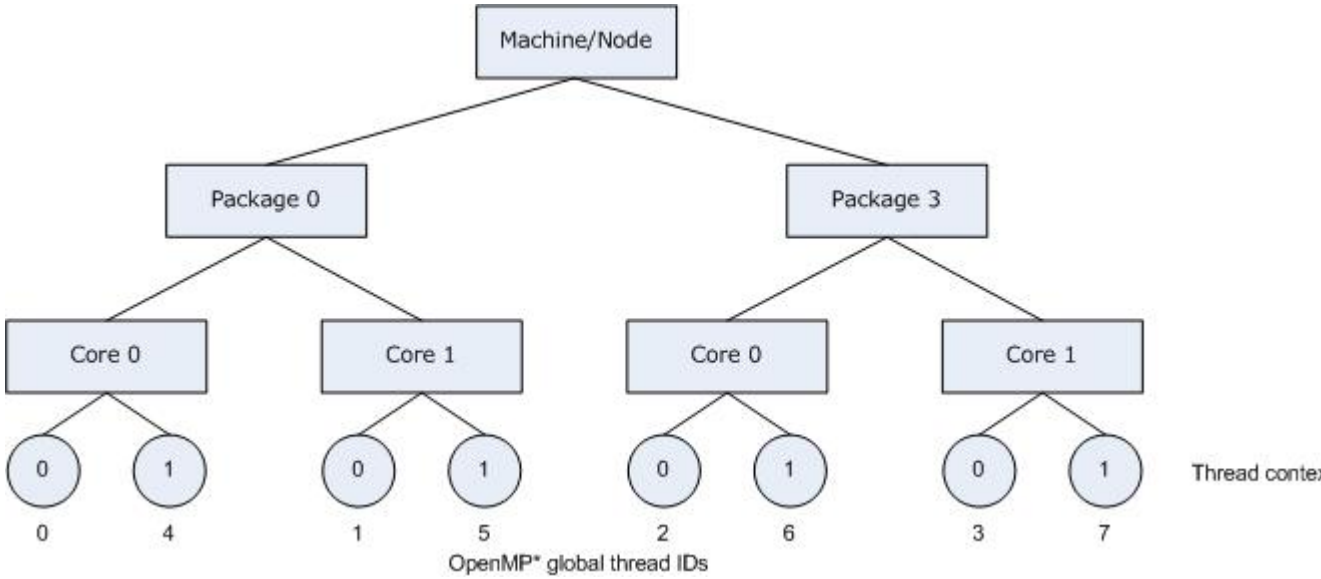
The `permute` specifier controls which levels are most significant when sorting the machine topology map. A value for permute forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The `offset` specifier indicates the starting position for thread assignment.

The following figure illustrates the result of specifying `KMP_AFFINITY=granularity=fine,compact,0,3`.



Consider the hardware configuration from the previous example, running an OpenMP application which exhibits data sharing between consecutive iterations of loops. We would therefore like consecutive threads to be bound close together, as is done with `KMP_AFFINITY=compact`, so that communication overhead, cache line invalidation overhead, and page thrashing are minimized. Now, suppose the application also had a number of parallel regions which did not utilize all of the available OpenMP threads. It is desirable to avoid binding multiple threads to the same core and leaving other cores not utilized, since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core. Since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core, you might want to avoid binding multiple threads to the same core while leaving other cores unused. The following figure illustrates this strategy of using `KMP_AFFINITY=granularity=fine,compact,1,0` as a setting.



The OpenMP thread *n*+1 is bound to a thread context as close as possible to OpenMP thread *n*, but on a different core. Once each core has been assigned one OpenMP thread, the subsequent OpenMP threads are assigned to the available cores in the same order, but they are assigned on different thread contexts.

Modifier Values for Affinity Types

Modifiers are optional arguments that precede type. If you do not specify a modifier, the `noverbose`, `respect`, and `granularity=core` modifiers are used automatically.

Modifiers are interpreted in order from left to right, and can negate each other. For example, specifying `KMP_AFFINITY=verbose,noverbose,scatter` is therefore equivalent to setting `KMP_AFFINITY=noverbose,scatter`, or just `KMP_AFFINITY=scatter`.

modifier = noverbose (default)

Does not print verbose messages.

modifier = verbose

Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP thread bindings to physical thread contexts.

Information about binding OpenMP threads to physical thread contexts is indirectly shown in the form of the mappings between hardware thread contexts and the operating system (OS) processor (proc) IDs. The affinity mask for each OpenMP thread is printed as a set of OS processor IDs.

For example, specifying `KMP_AFFINITY=verbose,scatter` on a dual core system with two processors, with Hyper-Threading Technology (HT Technology) disabled, results in a message listing similar to the following when then program is executed:

Verbose, scatter message

```
...
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}
```

The verbose modifier generates several standard, general messages. The following table summarizes how to read the messages.

Message String	Description
"affinity capable"	Indicates that all components (compiler, operating system, and hardware) support affinity, so thread binding is possible.

Message String	Description
"using global cpuid info"	Indicates that the machine topology was discovered by binding a thread to each operating system processor and decoding the output of the `cpuid` instruction.
"using local cpuid info"	Indicates that compiler is decoding the output of the `cpuid` instruction, issued by only the initial thread, and is assuming a machine topology using the number of operating system processors.
"using /proc/cpuinfo"	Linux* only. Indicates that `cpuinfo` is being used to determine machine topology.
"using flat"	Operating system processor ID is assumed to be equivalent to physical package ID. This method of determining machine topology is used if none of the other methods will work, but may not accurately detect the actual machine topology.
"uniform topology of"	The machine topology map is a full tree with no missing leaves at any level.

The mapping from the operating system processors to thread context ID is printed next. The binding of OpenMP thread context ID is printed next unless the affinity type is `none`. The thread level is contained in brackets (in the listing shown above). This implies that there is no representation of the thread context level in the machine topology map. For more information, see Determining Machine Topology.

modifier = granularity

Binding OpenMP threads to particular packages and cores will often result in a performance gain on systems with Intel processors with Hyper-Threading Technology (HT Technology) enabled; however, it is usually not beneficial to bind each OpenMP thread to a particular thread context on a specific core. Granularity describes the lowest levels that OpenMP threads are allowed to float within a topology map.

This modifier supports the following additional specifiers.

Specifier	Description
`core`	Default. Broadest granularity level supported. Allows all the OpenMP threads bound to a core to float between the different thread contexts.
`fine` or `thread`	The finest granularity level. Causes each OpenMP thread to be bound to a single thread context. The two specifiers are functionally equivalent.

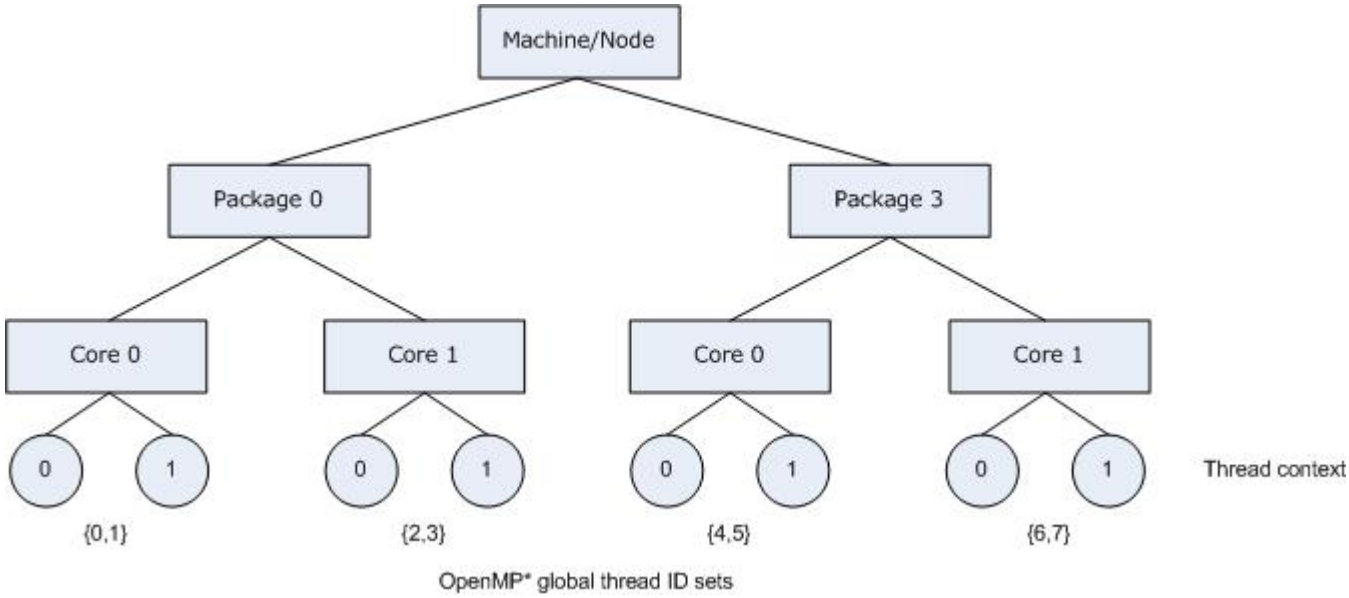
Specifying `KMP_AFFINITY=verbose,granularity=core,compact` on the same dual core system with two processors as in the previous section, but with HT Technology enabled, results in a message listing similar to the following when the program is executed:

Verbose, granularity=core,compact message

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,4}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0,4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {2,6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {1,5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3,7}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {3,7}
```

The affinity mask for each OpenMP thread is shown in the listing (above) as the set of operating system processor to which the OpenMP thread is bound.

The following figure illustrates the machine topology map, for the above listing, with OpenMP thread bindings.



In contrast, specifying `KMP_AFFINITY=verbose,granularity=fine,compact` or `KMP_AFFINITY=verbose,granularity=thread,compact` binds each OpenMP thread to a single hardware thread context when the program is executed:

Verbose, granularity=fine,compact message

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{0,1,2,3,4,5,6,7}
KMP_AFFINITY: 8 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 2 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 thread 1
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 thread 0
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 thread 1
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 thread 0
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 thread 1
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 thread 0
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 thread 1
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {1}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

The OpenMP to hardware context binding for this example was illustrated in the first example.

Specifying `granularity=fine` will always cause each OpenMP thread to be bound to a single OS processor. This is equivalent to `granularity=thread`, currently the finest granularity level.

modifier = respect (default)

Respect the process' original affinity mask, or more specifically, the affinity mask in place for the thread that initializes the OpenMP run-time library. The behavior differs between Linux and Windows OS:

- On Windows: Respect original affinity mask for the process.
- On Linux: Respect the affinity mask for the thread that initializes the OpenMP run-time library.

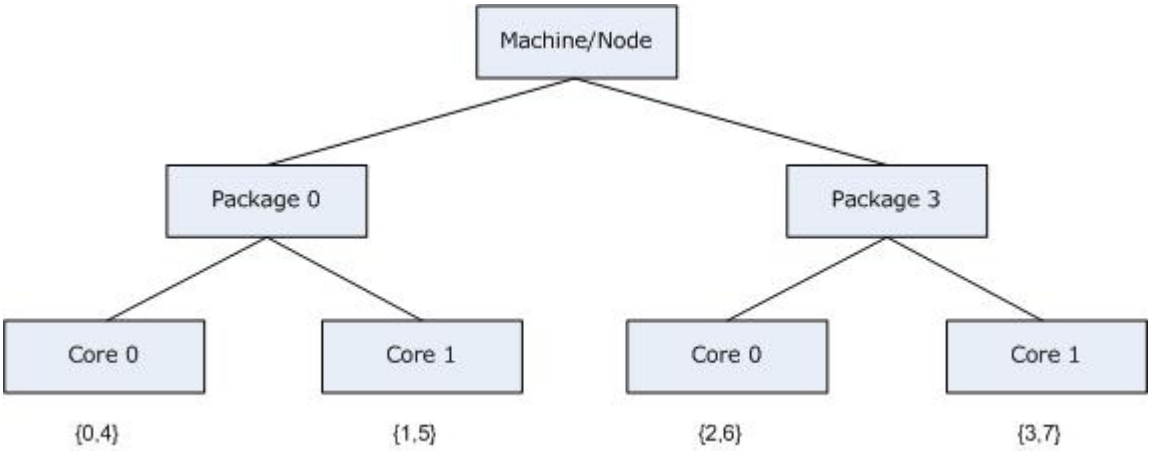
Specifying `KMP_AFFINITY=verbose,compact` for the same system used in the previous example, with HT Technology enabled, and invoking the library with an initial affinity mask of {4,5,6,7} (thread context 1 on every core) causes the compiler to model the machine as a dual core, two-processor system with HT Technology disabled.

Verbose,compact message

```
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected:
{4,5,6,7}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 4 maps to package 0 core 0 [thread 1]
KMP_AFFINITY: OS proc 6 maps to package 0 core 1 [thread 1]
KMP_AFFINITY: OS proc 5 maps to package 3 core 0 [thread 1]
KMP_AFFINITY: OS proc 7 maps to package 3 core 1 [thread 1]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {7}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {4}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {6}
KMP_AFFINITY: Internal thread 6 bound to OS proc set {5}
KMP_AFFINITY: Internal thread 7 bound to OS proc set {7}
```

Because there are eight thread contexts on the machine, by default the compiler created eight threads for an OpenMP `parallel` construct.

The brackets around thread 1 indicate that the thread context level is ignored, and is not present in the topology map. The following figure illustrates the corresponding machine topology map.



OpenMP* global thread ID sets

When using the local `cpuid` information to determine the machine topology, it is not always possible to distinguish between a machine that does not support Hyper-Threading Technology (HT Technology) and a machine that supports it, but has it disabled. Therefore, the compiler does not include a level in the map if the elements (nodes) at that level had no siblings, with the exception that the package level is always modeled. As mentioned earlier, the package level will always appear in the topology map, even if there only a single package in the machine.

modifier = norespect

Do not respect original affinity mask for the process. Binds OpenMP threads to all operating system processors.

In early versions of the OpenMP run-time library that supported only the `physical` and `logical` affinity types, `norespect` was the default and was not recognized as a modifier.

The default was changed to `respect` when types `compact` and `scatter` were added; therefore, thread bindings for the `logical` and `physical` affinity types may have changed with the newer compilers in situations where the application specified a partial initial thread affinity mask.

modifier = nowarnings

Do not print warning messages from the affinity interface.

modifier = warnings (default)

Print warning messages from the affinity interface (default).

Determining Machine Topology

On IA-32 and Intel® 64 architecture systems, if the package has an APIC (Advanced Programmable Interrupt Controller), the compiler will use the `cpuid` instruction to obtain the `package id`, `core id`, and `thread context id`. Under normal conditions, each thread context on the system is assigned a unique APIC ID at boot time. The compiler obtains other pieces of information obtained by using the *`cpuid`* instruction, which together with the number of OS thread contexts (total number of processing elements on the machine), determine how to break the APIC ID down into the `package ID`, `core ID`, and `thread context ID`.

Normally, all `core ids` on a package and all `thread context ids` on a core are contiguous; however, numbering assignment gaps are common for `package ids`, as shown in the figure above.

On IA-64 architecture systems on Linux* operating systems, the compiler obtains this information from `/proc/cpuinfo`. The `package id`, `core id`, and `thread context id` are obtained from the `physical id`, `core id`, and `thread id` fields from `/proc/cpuinfo`. The `core id` and `thread context id` default to 0, but the `physical id` field must be present in order to determine the machine topology, which is not always the case. If the information contained in `/proc/cpuinfo` is insufficient or erroneous, you may create an alternate specification file and pass it to the OpenMP runtime library by using the KMP_CPUINFO_FILE environment variable, as described in KMP_CPUINFO and /proc/cpuinfo.

If the compiler cannot determine the machine topology using either method, but the operating system supports affinity, a warning message is printed, and the topology is assumed to be `flat`. For example, a flat topology assumes the operating system process *`N`* maps to package *`N`*, and there exists only one thread context per core and only one core for each package. (This assumption is always the case for processors based on IA-64 architecture running Windows.)

If the machine topology cannot be accurately determined as described above, the user can manually copy `/proc/cpuinfo` to a temporary file, correct any errors, and specify the machine topology to the OpenMP runtime library via the environment variable `KMP_CPUINFO_FILE=<temp_filename>`, as described in the section KMP_CPUINFO_FILE and /proc/cpuinfo.

Regardless of the method used in determining the machine topology, if there is only one thread context per core for every core on the machine, the thread context level will not appear in the topology map. If there is only one core per package for every package in the machine, the core level will not appear in the machine topology map. The topology map need not be a full tree, because different packages may contain a different number of cores, and different cores may support a different number of thread contexts.

The package level will always appear in the topology map, even if there only a single package in the machine.

KMP_CPUINFO and /proc/cpuinfo

One of the methods the Intel compiler OpenMP runtime library can use to detect the machine topology on Linux* systems is to parse the contents of `/proc/cpuinfo`. If the contents of this file (or a device mapped into the Linux file system) are insufficient or erroneous, you can consider copying its contents to a writable temporary file <*`temp_file`*>,correct it or extend it with the necessary information, and set KMP_CPUINFO_FILE=<*`temp_file`*>.

If you do this, the OpenMP runtime library will read the <*`temp_file`*> location pointed to by KMP_CPUINFO_FILE instead of the information contained in `/proc/cpuinfo` or attempting to detect the machine topology by decoding the APIC IDs. That is, the information contained in the <*`temp_file`*> overrides these other methods. You can use the KMP_CPUINFO_FILE interface on Windows* systems, where `/proc/cpuinfo` does not exist.

The content of `/proc/cpuinfo` or <*`temp_file`*> should contain a list of entries for each processing element on the machine. Each processor element contains a list of entries (descriptive name and value on each line). A blank line separates the entries for each processor element. Only the following fields are used to determine the machine topology from each entry, either in <*`temp_file`*> or `/proc/cpuinfo`:

Field	Description
processor :	Specifies the OS ID for the processing element. The OS ID must be unique. The `processor` and `physical id` fields are the only ones that are required to use the interface.
physical id :	Specifies the package ID, which is a physical chip ID. Each package may contain multiple cores. The package level always exists in the Intel compiler OpenMP run-time library's model of the machine topology.
core id :	Specifies the core ID. If it does not exist, it defaults to 0. If every package on the machine contains only a single core, the core

Field	Description
	level will not exist in the machine topology map (even if some of the core ID fields are non-zero).
thread id :	Specifies the thread ID. If it does not exist, it defaults to 0. If every core on the machine contains only a single thread, the thread level will not exist in the machine topology map (even if some thread ID fields are non-zero).
node_*n* id :	This is a extension to the normal contents of `/proc/cpuinfo` that can be used to specify the nodes at different levels of the memory interconnect on Non-Uniform Memory Access (NUMA) systems. Arbitrarily many levels *n* are supported. The node_0 level is closest to the package level; multiple packages comprise a node at level 0. Multiple nodes at level 0 comprise a node at level 1, and so on.

Each entry must be spelled exactly as shown, in lowercase, followed by optional whitespace, a colon (:), more optional whitespace, then the integer ID. Fields other than those listed are simply ignored.

NOTE. It is common for the `thread id` field to be missing from `/proc/cpuinfo` on many Linux variants, and for a field labeled `siblings` to specify the number of threads per node or number of nodes per package. However, the Intel compiler OpenMP runtime library ignores fields labeled `siblings` so it can distinguish between the `thread id` and `siblings` fields. When this situation arises, the warning message `Physical node/pkg/core/thread ids not unique` appears (unless the `type specified is nowarnings`).

The following is a sample entry for an IA-64 architecture system that has been extended to model the different levels of the memory interconnect:

Sample `/proc/cpuinfo` **or <*temp-file*>**

```
processor : 23
vendor : GenuineIntel
arch : IA-64
family : 32
model : 0
revision : 7
archrev : 0
features : branchlong, 16-byte atomic ops
cpu number : 0
cpu regs : 4
cpu MHz : 1594.000007
itc MHz : 399.000000
BogoMIPS : 3186.68
siblings : 2
node_3 id : 0
node_2 id : 1
node_1 id : 0
node_0 id : 1
physical id : 2563
core id: 1
thread id: 0
```

This example includes the fields from `/proc/cpuinfo` that affect the functionality of the Intel compiler OpenMP Affinity Interface: `processor`, `physical id`, `core id`, and `thread id`. Other fields (`vendor`, `arch`, ..., `siblings`) from `/proc/cpuinfo` are ignored. The four fields `node_n` are extensions.

Explicitly Specifying OS Processor IDs (GOMP_CPU_AFFINITY)

Instead of allowing the library to detect the hardware topology and automatically assign OpenMP threads to processing elements, the user may explicitly specify the assignment by using a list of operating system (OS) processor (proc) IDs. However, this requires knowledge of which processing elements the OS proc IDs represent.

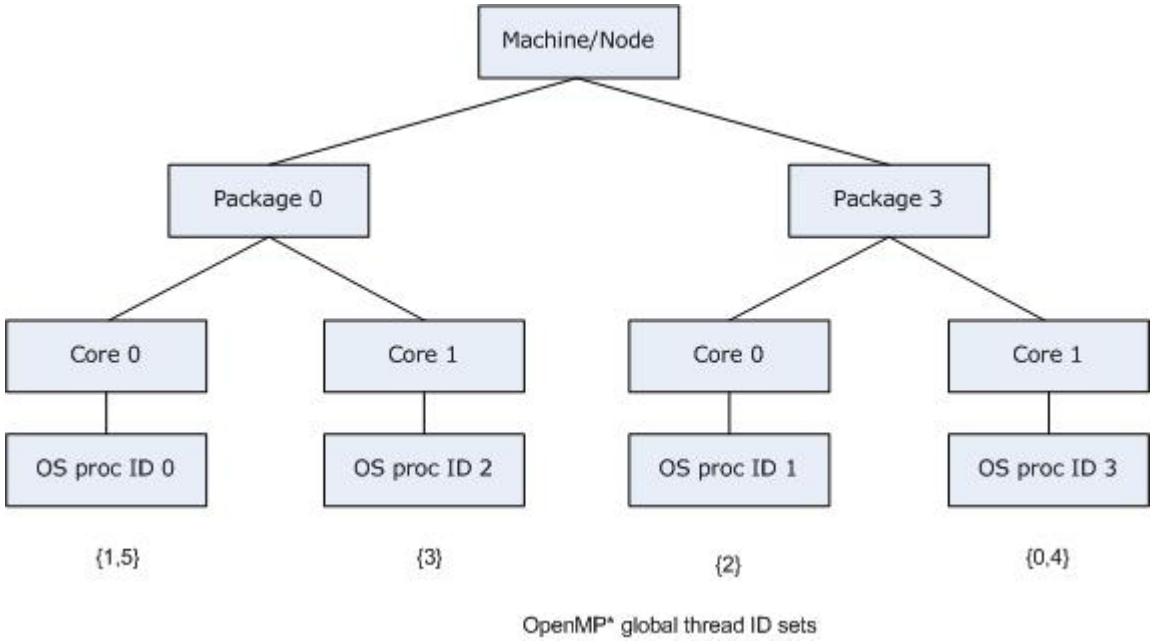
This list may either be specified by using the proclist modifier along with the explicit affinity type in the KMP_AFFINITY environment variable, or by using the GOMP_CPU_AFFINITY environment variable (for compatibility with `gcc`) when using the Intel OpenMP compatibility libraries.

On Linux systems when using the Intel OpenMP compatibility libraries enabled by the compiler option `-openmp-lib compat`, you can use the GOMP_CPU_AFFINITY environment variable to specify a list of OS processor IDs. Its syntax is identical to that accepted by `libgomp` (assume that <*proc_list*> produces the entire GOMP_CPU_AFFINITY environment string):

<*proc_list*> :=	<entry> | <*elem*> , <list> | <elem> <whitespace> <list>
<*elem*> :=	<*proc_spec*> | <*range*>
<*proc_spec*> :=	<*proc_id*>
<*range*> :=	<*proc_id*> - <*proc_id*> | <*proc_id*> - <*proc_id*> : <int>
<*proc_id*> :=	<positive_int>

OS processors specified in this list are then assigned to OpenMP threads, in order of OpenMP Global Thread IDs. If more OpenMP threads are created than there are elements in the list, then the assignment occurs modulo the size of the list. That is, OpenMP Global Thread ID *n* is bound to list element *n* mod <*list_size*>.

Consider the machine previously mentioned: a dual core, dual-package machine without Hyper-Threading Technology (HT Technology) enabled, where the OS proc IDs are assigned in the same manner as the example in a previous figure. Suppose that the application creates 6 OpenMP threads instead of 4 (the default), oversubscribing the machine. If `GOMP_CPU_AFFINITY=3,0-2`, then OpenMP threads are bound as shown in the figure below, just as should happen when compiling with `gcc` and linking with `libgomp`:



OpenMP* global thread ID sets

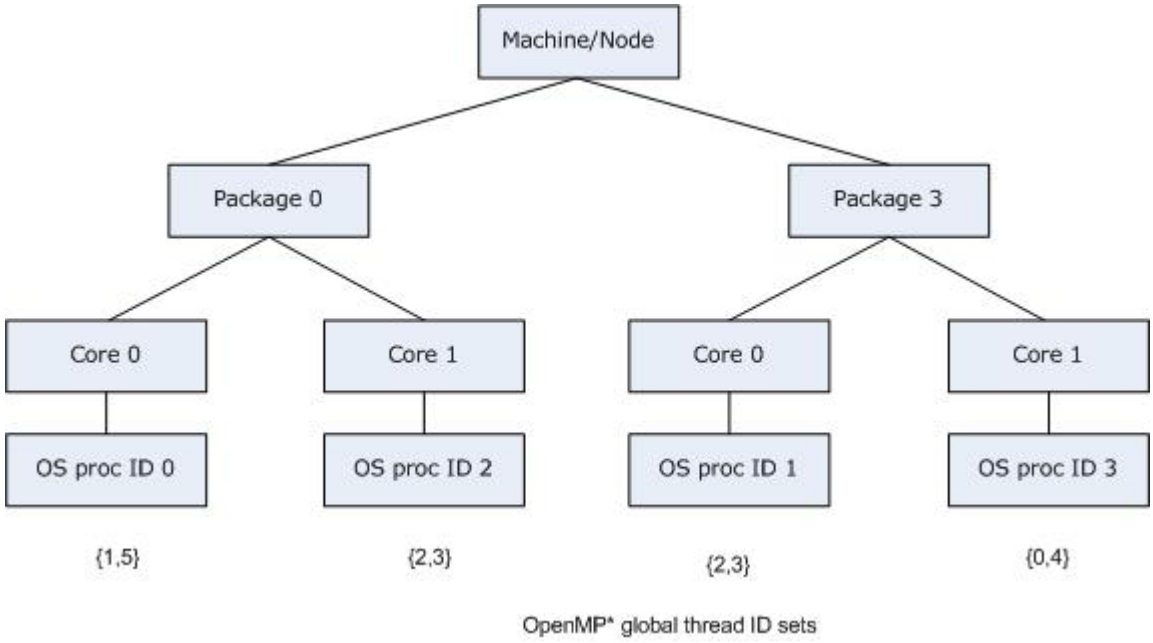
The same syntax can be used to specify the OS proc ID list in the proclist=[<proc_list>] modifier in the KMP_AFFINITY environment variable string. There is a slight difference: in order to have strictly the same semantics as in the `gcc` OpenMP runtime library `libgomp`: the GOMP_CPU_AFFINITY environment variable implies `granularity=fine`. If you specify the OS proc list in the KMP_AFFINITY environment variable without a `granularity=` specifier, then the default `granularity` is not changed. That is, OpenMP threads are allowed to float between the different thread contexts on a single core. Thus GOMP_CPU_AFFINITY=`<proc_list>` is an alias for KMP_AFFINITY=`granularity=fine,proclist=[<proc_list>],explicit`

In the KMP_AFFINITY environment variable string, the syntax is extended to handle operating system processor ID sets. The user may specify a set of operating system processor IDs among which an OpenMP thread may execute ("œfloat") enclosed in brackets:

<*proc_list*> :=	<proc_id> | { <float_list> }
<*float_list*> :=	<proc_id> | <proc_id> , <float_list>

This allows functionality similarity to the `granularity= specifier`, but it is more flexible. The OS processors on which an OpenMP thread executes may exclude other OS processors nearby in the machine topology, but include other distant OS processors. Building upon the

previous example, we may allow OpenMP threads 2 and 3 to "œfloat" between OS processor 1 and OS processor 2 by using KMP_AFFINITY="`granularity=fine,proclist=[3,0,{1,2},{1,2}],explicit`", as shown in the figure below:



If `verbose` were also specified, the output when the application is executed would include:

```
KMP_AFFINITY="granularity=verbose,fine,proclist=[3,0,{1,2},{1,2}],explicit"
KMP_AFFINITY: Affinity capable, using global cpuid info
KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3}
KMP_AFFINITY: 4 available OS procs - Uniform topology of
KMP_AFFINITY: 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
KMP_AFFINITY: OS proc to physical thread map ([] => level not in map):
KMP_AFFINITY: OS proc 0 maps to package 0 core 0 [thread 0]
KMP_AFFINITY: OS proc 2 maps to package 0 core 1 [thread 0]
KMP_AFFINITY: OS proc 1 maps to package 3 core 0 [thread 0]
KMP_AFFINITY: OS proc 3 maps to package 3 core 1 [thread 0]
KMP_AFFINITY: Internal thread 0 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {0}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 3 bound to OS proc set {1,2}
KMP_AFFINITY: Internal thread 4 bound to OS proc set {3}
KMP_AFFINITY: Internal thread 5 bound to OS proc set {0}}
```

Low Level Affinity API

Instead of relying on the user to specify the OpenMP thread to OS proc binding by setting an environment variable before program execution starts (or by using the `kmp_settings` interface before the first parallel region is reached), each OpenMP thread may determine the desired set of OS procs on which it is to execute and bind to them with the `kmp_set_affinity` API call.

The Fortran API interfaces follow, where the type name `kmp_affinity_mask_t` is defined in `omp.h` or `omp.mod`:

Syntax	Description
`integer function kmp_set_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	Sets the affinity mask for the current OpenMP thread to `mask`, where `mask` is a set of OS proc IDs that has been created using the API calls listed below, and the thread will only execute on OS procs in the set. Returns either a zero (0) upon success or a nonzero error code.
`integer kmp_get_affinity(mask) integer (kind=kmp_affinity_mask_kind) mask`	Retrieves the affinity mask for the current OpenMP thread, and stores it in `mask`, which must have previously been initialized with a call to `kmp_create_affinity_mask()`. Returns either a zero (0) upon success or a nonzero error code.
`integer function kmp_get_affinity_max_proc()`	Returns the maximum OS proc ID that is on the machine, plus 1. All OS proc IDs are guaranteed to be between 0 (inclusive) and `kmp_get_affinity_max_proc()` (exclusive).
`subroutine kmp_create_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	Allocates a new OpenMP thread affinity mask, and initializes `mask` to the empty set of OS procs. The implementation is free to use an object of `kmp_affinity_mask_t` either as the set itself, a pointer to the actual set, or an index into a table describing the set. Do not make any assumption as to what the actual representation is.
`subroutine kmp_destroy_affinity_mask(mask) integer (kind=kmp_affinity_mask_kind) mask`	Deallocates the OpenMP thread affinity mask. For each call to `kmp_create_affinity_mask()`, there should be a corresponding call to `kmp_destroy_affinity_mask()`.
`integer function kmp_set_affinity_mask_proc(proc, mask)`	Adds the OS proc ID proc to the set `mask`, if it is not already. Returns either a zero (0) upon success or a nonzero error code.

Syntax	Description

`integer proc`
`integer`
`(kind=kmp_affinity_mask_kind) mask`	

`integer function`
`kmp_unset_affinity_mask_proc(proc,`
`mask)`
`integer proc`
`integer`
`(kind=kmp_affinity_mask_kind) mask`
	If the OS proc ID proc is in the set `mask`, it removes it. Returns either a zero (0) upon success or a nonzero error code.

`integer function`
`kmp_get_affinity_mask_proc(proc,`
`mask)integer proc`
`integer`
`(kind=kmp_affinity_mask_kind) mask`
	Returns 1 if the OS proc ID proc is in the set `mask`; if not, it returns 0.

Once an OpenMP thread has set its own affinity mask via a successful call to `kmp_affinity_set_mask()`, then that thread remains bound to the corresponding OS proc set until at least the end of the parallel region, unless reset via a subsequent call to `kmp_affinity_set_mask()`.

Between parallel regions, the affinity mask (and the corresponding OpenMP thread to OS proc bindings) can be considered thread private data objects, and have the same persistence as described in the OpenMP Application Program Interface. For more information, see the OpenMP API specification (http://www.openmp.org), some relevant parts of which are provided below:

"In order for the affinity mask and thread binding to persist between two consecutive active parallel regions, all three of the following conditions must hold:

- Neither parallel region is nested inside another explicit parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the dyn-var internal control variable in the enclosing task region is false at entry to both parallel regions."

Therefore, by creating a parallel region at the start of the program whose sole purpose is to set the affinity mask for each thread, the user can mimic the behavior of the KMP_AFFINITY environment variable with low-level affinity API calls, if program execution obeys the three aforementioned rules from the OpenMP specification. Consider again the example presented

in the previous figure. To mimic KMP_AFFINITY=compact, in each OpenMP thread with global thread ID n, we need to create an affinity mask containing OS proc IDs n modulo c, n modulo c + c, and so on, where c is the number of cores. This can be accomplished by inserting the following C code fragment into the application that gets executed at program startup time:

Example

```
int main() {
#pragma omp parallel
{
int tmax = omp_get_max_threads();
int tnum = omp_get_thread_num();
int nproc = omp_get_num_procs();
int ncores = nproc / 2;
int i;
kmp_affinity_mask_t mask;
kmp_create_affinity_mask(&mask);
for (i = tnum % ncores; i < tmax; i += ncores) {
kmp_set_affinity_mask_proc(i, &mask);
}
if (kmp_set_affinity(&mask) != 0) <error>;
}
```

This program fragment was written with knowledge about the mapping of the OS proc IDs to the physical processing elements of the target machine. On another machine, or on the same machine with a different OS installed, the program would still run, but the OpenMP thread to physical processing element bindings could differ.

Using Parallelism: Automatic Parallelization

27

Auto-parallelization Overview

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multithreaded code. Automatic parallelization determines the loops that are good worksharing candidates, performs the dataflow analysis to verify correct parallel execution, and partitions the data for threaded code generation as needed in programming with OpenMP* directives. The OpenMP and auto-parallelization applications provide the performance gains from shared memory on multiprocessor and dual core systems.

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multithreaded code. The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multithreaded code for those loops which can safely and efficiently be executed in parallel.

This behavior enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems.

Automatic parallelization frees developers from having to:

- find loops that are good worksharing candidates
- perform the dataflow analysis to verify correct parallel execution
- partition the data for threaded code generation as is needed in programming with OpenMP* directives.

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

While OpenMP directives enable serial applications to transform into parallel applications quickly, a programmer must explicitly identify specific portions of the application code that contain parallelism and add the appropriate compiler directives.

Auto-parallelization, which is triggered by the `-parallel` (Linux* OS and Mac OS* X) or `/Qparallel` (Windows* OS) option, automatically identifies those loop structures that contain parallelism. During compilation, the compiler automatically attempts to deconstruct the code sequences into separate threads for parallel processing. No other effort by the programmer is needed.

NOTE. IA-64 architecture only: Specifying these options implies `-opt-mem-bandwith1` (Linux) or `/Qopt-mem-bandwidth1` (Windows).

Serial code can be divided so that the code can execute concurrently on multiple threads. For example, consider the following serial code example.

Example 1: Original Serial Code

```
subroutine ser(a, b, c)
  integer, dimension(100) :: a, b, c
  do i=1,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine ser
```

The following example illustrates one method showing how the loop iteration space, shown in the previous example, might be divided to execute on two threads.

Example 2: Transformed Parallel Code

```
subroutine par(a, b, c)
  integer, dimension(100) :: a, b, c
  ! Thread 1
  do i=1,50
    a(i) = a(i) + b(i) * c(i)
  enddo
  ! Thread 2
  do i=51,100
    a(i) = a(i) + b(i) * c(i)
  enddo
end subroutine par
```

Auto-Vectorization and Parallelization

Auto-vectorization detects low-level operations in the program that can be done in parallel, and then converts the sequential program to process 2, 4, 8 or up to 16 elements in one operation, depending on the data type. In some cases auto-parallelization and vectorization

can be combined for better performance results. For example, in the code below, thread-level parallelism can be exploited in the outermost loop, while instruction-level parallelism can be exploited in the innermost loop.

Example

```
DO I = 1, 100     ! Execute groups of iterations in different hreads (TLP)
  DO J = 1, 32    ! Execute in SIMD style with multimedia extension (ILP)
     A(J,I) = A(J,I) + 1
  ENDDO
ENDDO
```

Auto-vectorization can help improve performance of an application that runs on systems based on Pentium®, Pentium with MMX™ technology, Pentium II, Pentium III, and Pentium 4 processors.

With the right choice of options, you can:

- Increase the performance of your application with minimum effort
- Use compiler features to develop multithreaded programs faster

Additionally, with the relatively small effort of adding OpenMP directives to existing code you can transform a sequential program into a parallel program. The following example shows OpenMP directives within the code.

Example

```
!OMP$ PARALLEL PRIVATE(NUM), SHARED (X,A,B,C)
! Defines a parallel region
```

Example

```
!OMP$ PARALLEL DO
! Specifies a parallel region that
! implicitly contains a single DO directive
DO I = 1, 1000
  NUM = FOO(B(i), C(I))
  X(I) = BAR(A(I), NUM)
! Assume FOO and BAR have no other effect
ENDDO
```

See examples of the auto-parallelization and auto-vectorization directives in the following topics.

Auto-Parallelization Options Quick Reference

These options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* and Mac OS* X	Windows*	Description
-parallel	/Qparallel	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. IA-64 architecture only: • Implies -opt-mem-bandwidth1 (Linux) or /Qopt-mem-bandwidth1 (Windows). Depending on the program and level of parallelization desired, you might need to set the KMP_STACKSIZE environment variable to an appropriately large size.

Linux* and Mac OS* X	Windows*	Description
`-par-threshold[n]`	`/Qpar-threshold[:n]`	Sets a threshold for the auto of loops based on the probability of profitable execution of the loop in parallel; valid values of *n* can be 0 to 100.
`-par-schedule-keyword`	`/Qpar-schedule-keyword`	Specifies the scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.
`-par-report`	`/Qpar-report`	Controls the diagnostic levels in the auto-parallelizer optimizer.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Auto-parallelization: Enabling, Options, Directives, and Environment Variables

To enable the auto-parallelizer, use the `-parallel` (Linux* and Mac OS* X) or `/Qparallel` (Windows*) option. This option detects parallel loops capable of being executed safely in parallel and automatically generates multi-threaded code for these loops.

NOTE. You might need to set the `KMP_STACKSIZE` environment variable to an appropriately large size to enable parallelization with this option.

An example of the command using auto-parallelization is as follows:

Operating System	Description
Linux and Mac OS X	`ifort -c -parallel myprog.f`
Windows	`ifort -c /Qparallel myprog.f`

Auto-parallelization uses two specific directives, `!DEC$ PARALLEL` and `!DEC$ NO PARALLEL`.

Auto-parallelization Directives Format and Syntax

The format of an auto-parallelization compiler directive is:

Syntax

```
<prefix> <directive>
```

where the brackets above mean:

- <xxx>: the prefix and directive are required
  - For fixed form source input, the prefix is !DEC$ or CDEC$
  - For free form source input, the prefix is !DEC$ only

The prefix is followed by the directive name; for example:

Syntax

```
!DEC$ PARALLEL
```

Since auto-parallelization directives begin with an exclamation point, the directives take the form of comments if you omit the -parallel (Linux) or /Qparallel (Windows) option.

The !DEC$ PARALLEL directive instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the immediately following loop. However, if dependencies are proven, they are not ignored.

The !DEC$ NOPARALLEL directive disables auto-parallelization for the following loop:

Example

```
program main
parameter (n=100
integer x(n),a(n)

!DEC$ NOPARALLEL
do i=1,n
  x(i) = i
enddo
```

Example

```
!DEC$ PARALLEL
do i=1,n
  a( x(i) ) = i
enddo
end
```

Auto-parallelization Environment Variables

Auto-parallelization uses the following OpenMP* environment variables.

- `OMP_NUM_THREADS`
- `OMP_SCHEDULE`
- `KMP_STACKSIZE`

See OpenMP* Environment Variables for more information about the default settings and how to use these variables.

Programming with Auto-parallelization

The auto-parallelization feature implements some concepts of OpenMP*, such as the worksharing construct (with the PARALLEL DO directive) . See Programming with OpenMP for worksharing construct. This section provides details on auto-parallelization.

Guidelines for Effective Auto-parallelization Usage

A loop can be parallelized if it meets the following criteria:

- The loop is countable at compile time: this means that an expression representing how many times the loop will execute (also called "the loop trip count") can be generated just before entering the loop.
- There are no `FLOW` (`READ` after `WRITE`), `OUTPUT` (`WRITE` after `WRITE`) or `ANTI` (`WRITE` after `READ`) loop-carried data dependencies. A loop-carried data dependency occurs when the same memory location is referenced in different iterations of the loop. At the compiler's discretion, a loop may be parallelized if any assumed inhibiting loop-carried dependencies can be resolved by run-time dependency testing.

The compiler may generate a run-time test for the profitability of executing in parallel for loop with loop parameters that are not compile-time constants.

Coding Guidelines

Enhance the power and effectiveness of the auto-parallelizer by following these coding guidelines:

- Expose the trip count of loops whenever possible; specifically use constants where the trip count is known and save loop parameters in local variables.
- Avoid placing structures inside loop bodies that the compiler may assume to carry dependent data, for example, procedure calls, ambiguous indirect references or global references.
- Insert the `!DEC$ PARALLEL` directive to disambiguate assumed data dependencies.
- Insert the `!DEC$ NOPARALLEL` directive before loops known to have insufficient work to justify the overhead of sharing among threads.

Auto-parallelization Data Flow

For auto-parallelization processing, the compiler performs the following steps:

1. Data flow analysis: Computing the flow of data through the program.
2. Loop classification: Determining loop candidates for parallelization based on correctness and efficiency, as shown by threshold analysis.
3. Dependency analysis: Computing the dependency analysis for references in each loop nest.
4. High-level parallelization: Analyzing dependency graph to determine loops which can execute in parallel, and computing run-time dependency.
5. Data partitioning: Examining data reference and partition based on the following types of access: `SHARED`, `PRIVATE`, and `FIRSTPRIVATE`.
6. Multithreaded code generation: Modifying loop parameters, generating entry/exit per threaded task, and generating calls to parallel run-time routines for thread creation and synchronization.

Programming for Multithread Platform Consistency

For applications where most of the computation is carried out in simple loops, Intel compilers may be able to generate a multithreaded version automatically. This information applies to applications built for deployment on symmetric multiprocessors (`SMP`), systems with Hyper-Threading Technology (HT Technology) enabled, and dual core processor systems.

The compiler can analyze dataflow in loops to determine which loops can be safely and efficiently executed in parallel. Automatic parallelization can sometimes result in shorter execution times. Compiler enabled auto-parallelization can help reduce the time spent performing several common tasks:

- searching for loops that are good candidates for parallel execution
- performing dataflow analysis to verify correct parallel execution
- adding parallel compiler directives manually

Parallelization is subject to certain conditions, which are described in the next section. If `-openmp` and `-parallel` (Linux* and Mac OS* X) or `/Qopenmp` and `/Qparallel` (Windows*) are both specified on the same command line, the compiler will only attempt to parallelize those functions that do not contain OpenMP* directives.

The following program contains a loop with a high iteration count:

Example

```
subroutine no_dep
  parameter (n=100000000)
  real a, c(n)
  do i = 1, n
    a = 2 * i - 1
    c(i) = sqrt(a)
  enddo
  print*, n, c(1), c(n)
end subroutine no_dep
```

Dataflow analysis confirms that the loop does not contain data dependencies. The compiler will generate code that divides the iterations as evenly as possible among the threads at runtime. The number of threads defaults to the number of processors but can be set independently using the `OMP_NUM_THREADS` environment variable. The increase in parallel speed for a given loop depends on the amount of work, the load balance among threads, the overhead of thread creation and synchronization, etc., but generally will be less than the number of threads. For a whole program, speed increases depend on the ratio of parallel to serial computation.

For builds with separate compiling and linking steps, be sure to link the OpenMP* runtime library when using automatic parallelization. The easiest way to do this is to use the Intel® compiler driver for linking.

Parallelizing Loops

Three requirements must be met for the compiler to parallelize a loop.

1. The number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, usually cannot be made parallel.
2. There can be no jumps into or out of the loop.
3. The loop iterations must be independent.

In other words, correct results must not logically depend on the order in which the iterations are executed. There may, however, be slight variations in the accumulated rounding error, as, for example, when the same quantities are added in a different order. In some cases, such as summing an array or other uses of temporary scalars, the compiler may be able to remove an apparent dependency by a simple transformation.

Potential aliasing of pointers or array references is another common impediment to safe parallelization. Two pointers are aliased if both point to the same memory location. The compiler may not be able to determine whether two pointers or array references point to the same memory location. For example, if they depend on function arguments, run-time data, or the results of complex calculations. If the compiler cannot prove that pointers or array references are safe and that iterations are independent, the compiler will not parallelize the loop, except in limited cases when it is deemed worthwhile to generate alternative code paths to test explicitly for aliasing at run-time. If you know parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using the `!DIR$ PARALLEL` directive.

The compiler can only effectively analyze loops with a relatively simple structure. For example, the compiler cannot determine the thread safety of a loop containing external function calls because it does not know whether the function call might have side effects that introduce dependences. Fortran90 programmers can use the `PURE` attribute to assert that subroutines and functions contain no side effects. You can invoke interprocedural optimization with the `-ipo` (Linux* OS and Mac OS X) or `/Qipo` (Windows) compiler option. Using this option gives the compiler the opportunity to analyze the called function for side effects.

When the compiler is unable to parallelize automatically loops you know to be parallel use OpenMP*. OpenMP* is the preferred solution because you, as the developer, understand the code better than the compiler and can express parallelism at a coarser granularity. On the other hand, automatic parallelization can be effective for nested loops, such as those in a matrix multiply. Moderately coarse-grained parallelism results from threading of the outer loop, allowing the inner loops to be optimized for fine-grained parallelism using vectorization or software pipelining.

If a loop can be parallelized, it's not always the case that it should be parallelized. The compiler uses a threshold parameter to decide whether to parallelize a loop. The `-par-threshold` (Linux* OS and Mac OS X) or `/Qpar-threshold` (Windows) compiler option adjusts this behavior. The threshold ranges from 0 to 100, where 0 instructs the compiler to always parallelize a safe loop and 100 instructs the compiler to only parallelize those loops for which a performance gain is highly probable. Use the `-par-report` (Linux* OS and Mac OS X) or `/Qpar-report`

(Windows) compiler option to determine which loops were parallelized. The compiler will also report which loops could not be parallelized indicate a probably reason why it could not be parallelized. See Auto-parallelization: Threshold Control and Diagnostics for more information on the using these compiler options.

Because the compiler does not know the value of *k*, the compiler assumes the iterations depend on each other, for example if k equals -1, even if the actual case is otherwise. You can override the compiler inserting `!DEC$ parallel`:

Example

```
subroutine add(k, a, b)
  integer :: k
  real :: a(10000), b(10000)
  !$DEC parallel
  do i = 1, 10000
    a(i) = a(i+k) + b(i)
  end do
end subroutine add
```

As the developer, it's your responsibility to not call this function with a value of *k* that is less than 10000; passing a value less than 10000 could to incorrect results.

Thread Pooling

Thread pools offer an effective approach to managing threads. A thread pool is a group of threads waiting for work assignments. In this approach, threads are created once during an initialization step and terminated during a finalization step. This simplifies the control logic for checking for failures in thread creation midway through the application and amortizes the cost of thread creation over the entire application. Once created, the threads in the thread pool wait for work to become available. Other threads in the application assign tasks to the thread pool. Typically, this is a single thread called the thread manager or dispatcher. After completing the task, each thread returns to the thread pool to await further work. Depending upon the work assignment and thread pooling policies employed, it is possible to add new threads to the thread pool if the amount of work grows. This approach has the following benefits:

- Possible runtime failures midway through application execution due to inability to create threads can be avoided with simple control logic.
- Thread management costs from thread creation are minimized. This in turn leads to better response times for processing workloads and allows for multithreading of finer-grained workloads.

A typical usage scenario for thread pools is in server applications, which often launch a thread for every new request. A better strategy is to queue service requests for processing by an existing thread pool. A thread from the pool grabs a service request from the queue, processes it, and returns to the queue to get more work.

Thread pools can also be used to perform overlapping asynchronous I/O. The I/O completion ports provided with the Win32* API allow a pool of threads to wait on an I/O completion port and process packets from overlapped I/O operations.

OpenMP* is strictly a fork/join threading model. In some OpenMP implementations, threads are created at the start of a parallel region and destroyed at the end of the parallel region. OpenMP applications typically have several parallel regions with intervening serial regions. Creating and destroying threads for each parallel region can result in significant system overhead, especially if a parallel region is inside a loop; therefore, the Intel OpenMP implementation uses thread pools. A pool of worker threads is created at the first parallel region. These threads exist for the duration of program execution. More threads may be added automatically if requested by the program. The threads are not destroyed until the last parallel region is executed.

Thread pools can be created on Windows and Linux using the thread creation API.

The function `CheckPoolQueue` executed by each thread in the pool is designed to enter a wait state until work is available on the queue. The thread manager can keep track of pending jobs in the queue and dynamically increase the number of threads in the pool based on the demand.

Using Parallelism: Automatic Vectorization

28

Automatic Vectorization Overview

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in the MMX™, Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators) and Supplemental Streaming SIMD Extensions (SSSE3) instruction sets. The vectorizer detects operations in the program that can be done in parallel, and then converts the sequential operations like one SIMD instruction that processes 2, 4, 8 or up to 16 elements in parallel, depending on the data type.

Automatic vectorization is supported on IA-32 and Intel® 64 architectures.

The section discusses the following topics, among others:

- High-level discussion of compiler options used to control or influence vectorization
- Vectorization Key Programming Guidelines
- Loop parallelization and vectorization
- Discussion and general guidelines on vectorization levels:
  - automatic vectorization
  - vectorization with user intervention
- Examples demonstrating typical vectorization issues and resolutions

The compiler supports a variety of directives that can help the compiler to generate effective vector instructions.

See Vectorization Support.

See *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*, A.J.C. Bik. Intel Press, June, 2004, for a detailed discussion of how to vectorize code using the Intel® compiler. Additionally, see the Related Publications topic in this document for other resources.

Automatic Vectorization Options Quick Reference

These options are supported on IA-32 and Intel® 64 architectures.

Linux* OS and Mac OS* X	Windows* OS	Description
-x	/Qx	Generates specialized code to run exclusively on processors with the extensions specified as the processor value. See Targeting IA-32 and Intel® 64 Architecture Processors Automatically for more information about using the option.
-ax	/Qax	Generates, in a single binary, code specialized to the extensions specified as the processor value and also generic IA-32 architecture code. The generic code is usually slower. See Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance for more information about using the option.
-vec	/Qvec	Enables or disables vectorization and transformations enabled for vectorization. The default is that vectorization is enabled. Supported for IA-32 and Intel® 64 architectures only.
-vec-report	/Qvec-report	Controls the diagnostic messages from the vectorizer. See Vectorization Report.

Vectorization within the Intel® compiler depends upon ability of the compiler to disambiguate memory references. Certain options may enable the compiler to do better vectorization.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Programming Guidelines for Vectorization

The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help however by supplying the compiler with additional information; for example, by using directives.

Guidelines

You will often need to make some changes to your loops. Follow these guidelines for loop bodies.

Use:

- straight-line code (a single basic block)
- vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- only assignment statements

Avoid:

- function calls
- unvectorizable operations (other than mathematical)
- mixing vectorizable types in the same loop
- data-dependent loop exit conditions

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- loop unrolling; the compiler does it automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

Restrictions

There are a number of restrictions that you should be consider. Vectorization depends on two major factors: hardware and style of source code.

Factor	Description
Hardware	The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture.
Style of source code	The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Many stylistic issues that prevent automatic vectorization by compilers are found in loop structures. The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the known limitations and enable effective vectorization. The following sections summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

Vectorization and Loops

Combine the `-parallel` (Linux* and Mac OS* X) or `/Qparallel` (Windows*) and `-x` (Linux) or `/Qx` (Windows) options to instructs the compiler to attempt both automatic loop parallelization and automatic loop vectorization in the same compilation.

In most cases, the compiler will consider outermost loops for parallelization and innermost loops for vectorization. If deemed profitable, however, the compiler may even apply loop parallelization and vectorization to the same loop.

See Guidelines for Effective Auto-parallelization Usage and Programming Guidelines for Vectorization.

In some rare cases successful loop parallelization (either automatically or by means of OpenMP* directives) may affect the messages reported by the compiler for a non-vectorizable loop in a non-intuitive way; for example, in the cases where `-vec-report2` (Linux and Mac OS X) or `/Qvec-report2` (Windows) option indicating loops were not successfully vectorized. (See Vectorization Report.)

Types of Vectorized Loops

For integer loops, the 64-bit MMX™ technology and 128-bit Intel® Streaming SIMD Extensions (Intel® SSE) provide SIMD instructions for most arithmetic and logical operators on 32-bit, 16-bit, and 8-bit integer data types.

Vectorization may proceed if the final precision of integer wrap-around arithmetic will be preserved. A 32-bit shift-right operator, for instance, is not vectorized in 16-bit mode if the final stored value is a 16-bit integer. Also, note that because the MMX™ and Intel® SSE instruction sets are not fully orthogonal (shifts on byte operands, for instance, are not supported), not all integer operations can actually be vectorized.

For loops that operate on 32-bit single-precision and 64-bit double-precision floating-point numbers, Intel® SSE provides SIMD instructions for the following arithmetic operators: addition (+), subtraction (-), multiplication (*), and division (/).

Additionally, the Streaming SIMD Extensions provide SIMD instructions for the binary `MIN` and `MAX` and unary `SQRT` operators. SIMD versions of several other mathematical operators (like the trigonometric functions `SIN`, `COS`, and `TAN`) are supported in software in a vector mathematical run-time library that is provided with the Intel® compiler of which the compiler takes advantage.

Statements in the Loop Body

The vectorizable operations are different for floating-point and integer data.

Integer Array Operations

The statements within the loop body may be arithmetic or logical operations (again, typically for arrays). Arithmetic operations are limited to such operations as addition, subtraction, `ABS`, `MIN`, and `MAX`. Logical operations include bitwise `AND`, `OR`, and `XOR` operators. You can mix data types only if the conversion can be done without a loss of precision. Some example operators where you can mix data types are multiplication, shift, or unary operators.

Other Operations

No statements other than the preceding floating-point and integer operations are valid. The loop body cannot contain any function calls other than the ones described above.

Data Dependency

Data dependency relations represent the required ordering constraints on the operations in serial loops. Because vectorization rearranges the order in which operations are executed, any auto-vectorizer must have at its disposal some form of data dependency analysis.

An example where data dependencies prohibit vectorization is shown below. In this example, the value of each element of an array is dependent on the value of its neighbor that was computed in the previous iteration.

Example 1: Data-dependent Loop

```
subroutine dep(data, n)
  real :: data(n)
  integer :: i
 do i = 1, n-1
    data(i) = data(i-1)*0.25 + data(i)*0.5 + data(i+1)*0.25
  end do
end subroutine dep
```

The loop in the above example is not vectorizable because the `WRITE` to the current element `DATA(I)` is dependent on the use of the preceding element `DATA(I-1)`, which has already been written to and changed in the previous iteration. To see this, look at the access patterns of the array for the first two iterations as shown below.

Example 2: Data-dependency Vectorization Patterns

```
I=1: READ DATA(0)
READ DATA(1)
READ DATA(2)
WRITE DATA(1)
I=2: READ DATA(1)
READ DATA(2)
READ DATA(3)
WRITE DATA(2)
```

In the normal sequential version of this loop, the value of `DATA(1)` read from during the second iteration was written to in the first iteration. For vectorization, it must be possible to do the iterations in parallel, without changing the semantics of the original loop.

Data dependency Analysis

Data dependency analysis involves finding the conditions under which two memory accesses may overlap. Given two references in a program, the conditions are defined by:

- whether the referenced variables may be aliases for the same (or overlapping) regions in memory
- for array references, the relationship between the subscripts

For IA-32 architecture, data dependency analyzer for array references is organized as a series of tests, which progressively increase in power as well as in time and space costs.

First, a number of simple tests are performed in a dimension-by-dimension manner, since independency in any dimension will exclude any dependency relationship. Multidimensional arrays references that may cross their declared dimension boundaries can be converted to their linearized form before the tests are applied.

Some of the simple tests that can be used are the fast greatest common divisor (GCD) test and the extended bounds test. The GCD test proves independency if the GCD of the coefficients of loop indices cannot evenly divide the constant term. The extended bounds test checks for potential overlap of the extreme values in subscript expressions.

If all simple tests fail to prove independency, the compiler will eventually resort to a powerful hierarchical dependency solver that uses Fourier-Motzkin elimination to solve the data dependency problem in all dimensions.

Loop Constructs

Loops can be formed with the usual `DO-END DO` and `DO WHILE`, or by using an `IF`/`GOTO` and a label. The loops must have a single entry and a single exit to be vectorized. The following examples illustrate loop constructs that can and cannot be vectorized.

Example: Vectorizable structure

```
subroutine vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
      if (a(i) .lt. 0.0) a(i) = 0.0
        i = i + 1
  enddo
end subroutine vec
```

The following example shows a loop that cannot be vectorized because of the inherent potential for an early exit from the loop.

Example: Non-vectorizable structure

```
subroutine no_vec(a, b, c)
  dimension a(100), b(100), c(100)
  integer i
  i = 1
  do while (i .le. 100)
    a(i) = b(i) * c(i)
! The next statement allows early
!
exit from the loop and prevents
! vectorization of the loop.
    if (a(i) .lt. 0.0) go to 10
    i = i + 1
  enddo
  10 continue
end subroutine no_vecN
END
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations a loop executes. For example, fixed indexes for loops determine the iterations. The loop iterations must be countable; in other words, the number of iterations must be expressed as one of the following:

- A constant
- A loop invariant term
- A linear function of outermost loop indices

In the case where a loops exit depends on computation, the loops are not countable. The examples below show loop constructs that are countable and non-countable.

Example: Countable Loop

```
subroutine cnt1 (a, b, c, n, lb)
  dimension a(n), b(n), c(n)
  integer n, lb, i, count
! Number of iterations is "n - lb + 1"
  count = n
  do while (count .ge. lb)
    a(i) = b(i) * c(i)
    count = count - 1
    i = i + 1
  enddo ! lb is not defined within loop
end
```

The following example demonstrates a different countable loop construct.

Example: Countable Loop

```
! Number of iterations is (n-m+2)/2
subroutine cnt2 (a, b, c, m, n)
  dimension a(n), b(n), c(n)
  integer i, l, m, n
  i = 1;
  do l = m,n,2
    a(i) = b(i) * c(i)
    i = i + 1
  enddo
end
```

The following examples demonstrates a loop construct that is non-countable due to dependency loop variant count value.

Example: Non-Countable Loop

```
! Number of iterations is dependent on a(i)
subroutine foo (a, b, c)
  dimension a(100),b(100),c(100)
  integer i
  i = 1
  do while (a(i) .gt. 0.0)
    a(i) = b(i) * c(i)
    i = i + 1
  enddo
end
```

Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. For example, assume the compiler attempts to strip-mine the following loop:

Example1: Before Vectorization

```
i = 1
do while (i<=n)
  a(i) = b(i) + c(i) ! Original loop code
  i = i + 1
end do
```

The compiler might handle the strip mining and loop cleaning by restructuring the loop in the following manner:

Example 2: After Vectorization

```
!The vectorizer generates the following two loops
i = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
  a(i:i+3) = b(i:i+3) + c(i:i+3)
  i = i + 4
end do
do while (i <= n)
  a(i) = b(i) + c(i)     !Scalar clean-up loop
  i = i + 1
end do
```

Loop Blocking

It is possible to treat loop blocking as strip-mining in two or more dimensions. Loop blocking is a useful technique for memory performance optimization. The main purpose of loop blocking is to eliminate as many cache misses as possible. This technique transforms the memory domain

into smaller chunks rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

Consider the following example. The two-dimensional array *A* is referenced in the *j* (column) direction and then in the *i* (row) direction (column-major order); array *B* is referenced in the opposite manner (row-major order). Assume the memory layout is in column-major order; therefore, the access strides of array *A* and *B* for the code would be 1 and MAX, respectively. In example 2: `BS = block_size;` MAX must be evenly divisible by *BS*.

Consider the following loop example code:

Example: Original loop

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX
  DO J = 1, MAX
    A(I,J) = A(I,J) + B(J,I)
  ENDDO
ENDDO
```

The arrays could be blocked into smaller chunks so that the total combined size of the two blocked chunks is smaller than the cache size, which can improve data reuse. One possible way of doing this is demonstrated below:

Example: Transformed Loop after blocking

```
REAL A(MAX,MAX), B(MAX,MAX)
DO I =1, MAX, BS
  DO J = 1, MAX, BS
   DO II = I, I+MAX, BS-1
    DO J = J, J+MAX, BS-1
     A(II,JJ) = A(II,JJ) + B(JJ,II
    ENDDO
   ENDDO
  ENDDO
ENDDO
```

Loop Interchange and Subscripts: Matrix Multiply

Loop interchange need unit-stride constructs to be vectorized. Matrix multiplication is commonly written as shown in the following example:

Example: Typical Matrix Multiplication

```
subroutine matmul_slow(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do i = 1, n
    do j = 1, n
      do k = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j);
      end do
    end do
  end do
end subroutine matmul_slow
```

The use of `B(K,J)` is not a stride-1 reference and therefore will not normally be vectorizable.

If the loops are interchanged, however, all the references will become stride-1 as shown in the following example.

Example: Matrix Multiplication with Stride-1

```
subroutine matmul_fast(a, b, c)
  integer :: i, j, k
  real :: a(100,100), b(100,100), c(100,100)
  do j = 1, n
    do k = 1, n
      do i = 1, n
        c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
    enddo
  enddo
end subroutine matmul_fast
```

Interchanging is not always possible because of dependencies, which can lead to different results.

Absence of Loop-carried Memory Dependency with IVDEP Directive

For applications designed to run on IA-64 architectures, the `-ivdep-parallel` (Linux*) or `/Qivdep-parallel` (Windows*) option indicates there is no loop-carried memory dependency in the loop where an `ivdep` directive is specified. This technique is useful for some sparse matrix applications.

NOTE. Mac OS* X: This option is not supported.

For example, the following loop requires the parallel option in addition to the `ivdep` directive to ensure there is no loop-carried dependency for the store into `a()`.

Example

```
!DEC$ IVDEP
do j=1,n
  a(b(j)) = a(b(j))+1
enddo
```

Vectorization Examples

This section contains simple examples of some common issues in vector programming.

Argument Aliasing: A Vector Copy

The loop in the example of a vector copy operation does not vectorize because the compiler cannot prove that `DEST(A(I))` and `DEST(B(I))` are distinct.

Example: Unvectorizable Copy Due to Unproven Distinction

```
SUBROUTINE VEC_COPY(DEST,A,B,LEN)
DIMENSION DEST(*)
INTEGER A(*), B(*)
INTEGER LEN, I
  DO I=1,LEN
  DEST(A(I)) = DEST(B(I))
END DO
RETURN
END
```

Data Alignment

A 16-byte (Linux* and Mac OS* X) or 64-byte (Windows*) or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned in a way that its base address is a multiple of 16 (Linux and Mac OS X) or 32 (Windows).

The figure (below) shows the effect of a data cache unit (DCU) split due to misaligned data. The code loads the misaligned data across a 16-byte boundary, which results in an additional memory access causing a six- to twelve-cycle stall. You can avoid the stalls if you know that the data is aligned and you specify to assume alignment.

Figure 20: Misaligned Data Crossing 16- Byte Boundary

After vectorization, the loop is executed as shown in figure below.

Figure 21: Vector and Scalar Clean-up iterations

Both the vector iterations `A(1:4) = B(1:4);` and `A(5:8) = B(5:8);` can be implemented with aligned moves if both the elements `A(1)` and `B(1)` are 16-byte aligned.

CAUTION. If you use the vectorizer with incorrect alignment options the compiler will generate code with unexpected behavior. Specifically, using aligned moves on unaligned data, will result in an illegal instruction exception.

Alignment Strategy

The compiler has at its disposal several alignment strategies in case the alignment of data structures is not known at compile-time. A simple example is shown below (several other strategies are supported as well). If in the loop shown below the alignment of A is unknown, the compiler will generate a prelude loop that iterates until the array reference, that occurs the most, hits an aligned address. This makes the alignment properties of A known, and the vector loop is optimized accordingly. In this case, the vectorizer applies dynamic loop peeling, a specific Intel® Fortran feature.

Examples of Data Alignment

Example: Original loop

```
SUBROUTINE SIMLOOP(A)
REAL A(100)    ! alignment of argument A is unknown
DO I = 1, 100
  A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

Example: Aligning Data

```
! The vectorizer applies dynamic loop peeling as follows:
SUBROUTINE SIMLOOP(A)
REAL A(100)
! let P be (A%16)where A is address of A(1)
IF (P .NE. 0) THEN
  P = (16 - P)/4   ! determine run-time peeling factor
  DO I = 1, P
    A(I) = A(I) + 1.0
  ENDDO
ENDIF
! Now this loop starts at a 16-byte boundary, and will be
! vectorized accordingly
DO I = P + 1, 100
  A(I) = A(I) + 1.0
ENDDO
END SUBROUTINE
```

Using Parallelism: Multi-Threaded Applications

29

Creating Multithread Applications Overview

Intel® Fortran provides support for creating multithread applications. You should consider using more than one thread if your application needs to manage multiple activities, such as simultaneous keyboard input and calculations. One thread can process keyboard input while a second thread performs data transformation calculations. A third thread can update the display screen based on data from the keyboard thread. At the same time, other threads can access disk files, or get data from a communications port.

When using a multiprocessor machine (sometimes called an "SMP machine") you can achieve a substantial speedup on numerically intensive problems by dividing the work among different threads; the operating system will assign the different threads to different processors (symmetric multiprocessing or parallel execution). Even if you have a single-processor machine, multiple-window applications might benefit from multithreading because threads can be associated with different windows; one thread can be calculating while another is waiting for input.

While you might gain execution speed by having a program executed in multiple threads, there is overhead involved in managing the threads. You need to evaluate the requirements of your project to determine whether you should run it with more than one thread.

If your multithreaded code calls functions from the run-time library or does input/output, you must also link your code to the multithreaded version of the run-time libraries instead of the regular single-threaded ones.

See Also

- Using Parallelism: Multi-Threaded Applications
- Basic Concepts of Multithreading
- Writing a Multithread Program

Basic Concepts of Multithreading

A *thread* is a path of execution through a program. It is an executable entity that belongs to one and only one process. Each process has at least one thread of execution, automatically created when the process is created. Your main program runs in the first thread. A Windows thread consists of a stack, the state of the CPU registers, a security context, and an entry in the execution list of the system scheduler. Each thread shares all of the process's resources.

A *process* consists of one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores (a method of interthread communication), and dynamically allocated memory. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads should run and when they should run. Threads of lower priority might need to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to balance the CPU load.

Because threads require less system overhead and are easier to create than an entire process, they are useful for time- or resource-intensive operations that can be performed concurrently with other tasks. Threads can be used for operations such as background printing, monitoring a device for input, or backing up data while it is being edited.

When threads, processes, files, and communications devices are opened, the function that creates them returns a *handle*. Each handle has an associated Access Control List (ACL) that is used to check the security credentials of the process. Processes and threads can inherit a handle or give one away using functions described in this section. Objects and handles regulate access to system resources.

All threads in a process execute independently of one another. Unless you take special steps to make them communicate with each other, each thread operates while completely unaware of the existence of other threads in a process. Threads sharing common resources must coordinate their work by using semaphores or another method of interthread communication. For more information on interthread communication, see Sharing Resources.

Developing Multithread Applications

Writing a Multithread Program Overview

Multiple threads are best used for:

- Background tasks such as data calculations, database queries, and input gathering, which do not directly involve window management or user interface.
- Operations that are independent from one another that can benefit from concurrent processing.
- Asynchronous tasks such as polling on a serial port.

If your application contains tasks that require a private address space and private resources, you can protect them from the activities of other threads by creating multiple processes rather than multiple threads. See Working with Multiple Processes.

The sections that follow discuss the steps you need to consider in creating a multithread application:

- Modules for Multithread Programs

- Starting and Stopping Threads
- Thread Routine Format
- Sharing Resources
- Thread Local Storage (TLS)
- Synchronizing Threads
- Handling Errors in Multithread Programs
- Working with Multiple Processes
- Table of Multithread Routines

Modules for Multithread Programs

A module called `IFMT.MOD` is supplied with Intel® Visual Fortran.

It contains interface statements to the underlying Windows API routines as well as parameter and structure definitions used by the routines. You need to include a **`USE IFMT`** statement in the declarations section of every Fortran program unit (program, subroutine, function, or module) that uses multithread API routines.

The source code for the IFMT module (`IFMT.F90`) contains type definitions and external function declarations. You can use it as an added reference for the calling syntax, number, and type of arguments for a multithread procedure.

Other Windows APIs that support multithreading tasks (such as window management functions) are included in the `IFWIN.F90` module, available to your programs with the **`USE IFWIN`** statement.

For information about creating a Fortran Windows application, see Creating Windows Applications in Building Applications.

Starting and Stopping Threads

When you add threads to a process, you need to consider the costs to your process. Create only the number of threads that help your application respond and perform better. You can save time by multitasking, but remember that additional CPU time is needed to keep track of multiple threads. When you are deciding how many threads to create, you also need to consider what data can be process-specific, and what data is thread-specific. Sharing Resources discusses synchronizing access to variables and data.

One single call to the CreateThread function creates a thread, specifies security attributes and memory stack size, and names the routine for the thread to run. Windows allocates memory for the thread stack in the virtual address space of the application that contains the thread. Once a thread has finished processing, the CloseHandle routine frees the resources used by the thread.

Starting Threads

The `CreateThread` function creates a new thread. Its return value is an INTEGER(4) thread handle, used in communicating to the thread and when closing it. The syntax for this function is:

`CreateThread (`*`security`*`, `*`stack`*`, `*`thread_func`*`, `*`argument`*`, `*`flags`*`, `*`thread_id`*`)`

All arguments are INTEGER(4) variables except for *`thread_func`*, which names the routine for `CreateThread` to run. The arguments are as follows:

Argument	Description
`security`	This argument uses the `SECURITY_ATTRIBUTES` type, defined in `IFMT.F90`. If *`security`* is zero, the thread has the default security attributes of the parent process. For more information about setting security attributes for processes and threads, see the Platform SDK online reference.
`stack`	Defines the stack size of the new thread. All of an application's default stack space is allocated to the first thread of execution. As a result, you must specify how much memory to allocate for a separate stack for each additional thread your program needs. The `CreateThread` call allows you to specify the value for the stack size on each thread you create. A value of zero indicates the stack has the same size as the application's primary thread. The size of the stack is increased dynamically, if necessary, up to a limit of 1 MB.
`thread_func`	The starting address for the thread function.
`argument`	An optional argument for *`thread_func`*. Your program defines this parameter and how it is used.
`flags`	This argument lets you create a thread that will not begin processing until you signal it. The *`flags`* argument can take either of two

Argument	Description
	values: 0 or CREATE_SUSPENDED. If you specify 0, the thread is created and runs immediately after creation. If you specify CREATE_SUSPENDED, the thread is created, but does not run until you call the ResumeThread function.
thread_id	This argument is returned by CreateThread. It is a unique identifier for the thread, which you can use when calling other multithread routines. While the thread is running, no other thread has the same identifier. However, the operating system may use the identifier again for other threads once this one has completed. A thread can be referred to by its handle as well as its unique thread identifier. Synchronization functions such as WaitForSingleObject and WaitForMultipleObjects take the thread handle as an argument.

Stopping Threads

The ExitThread routine allows a thread to stop its own execution. The syntax is: CALL EXITTHREAD ([*Termination Status*])

Termination status may be queried by another thread. A termination status of 0 indicates normal termination. You can assign other termination status values and their meaning in your program.

When the called thread is no longer needed, the calling thread needs to close the handle for the thread. Use the CloseHandle routine to free memory used by the thread. A thread object is not deleted until the last thread handle is closed.

It is possible for more than one handle to be open to a thread: for example, if a program creates two threads, one of which waits for information from the other. In this case, two handles are open to the first thread: one from the thread requesting information, the other from the thread that created it. All handles are closed implicitly when the enclosing process terminates.

The TerminateThread routine allows one thread to terminate another, if the security attributes are set appropriately for both threads. DLLs attached to the thread are not notified that the thread is terminating, and its initial stack is not deallocated. Use Terminate Thread for emergencies only.

Other Thread Support Functions

Scheduling thread priorities is supported through the functions `GetThreadPriority` and `SetThreadPriority`. Use the priority class of a thread to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. If you need to manipulate priorities, be very careful not to give a thread too high a priority, or it can consume all of the available CPU time. A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using `REALTIME_PRIORITY_CLASS` may cause disk caches to not flush, hang the mouse, and so on.

When communicating with other threads, a thread uses a *`pseudohandle`* to refer to itself. A pseudohandle is a special constant that is interpreted as the current thread handle. Pseudohandles are only valid for the calling thread; they cannot be inherited by other threads. The `GetCurrentThread` function returns a pseudohandle for the current thread. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudohandles are not inherited.

To get the thread's identifier, use the `GetCurrentThreadId` function. The identifier uniquely identifies the thread in the system until it terminates. You can use the identifier to specify the thread itself whenever an identifier is required.

Use `GetExitCodeThread` to find out if a thread is still active, or if it is not, to find its exit status. Call `GetLastError` for more detailed information on the exit status. If one routine depends on a task being performed by a different thread, use the wait functions described in Synchronizing Threads instead of `GetExitCodeThread`.

Thread Routine Format

A thread routine is a function that runs in a separate thread from the main program and takes a single argument. In the first interface below, the argument is the address of a integer(4) variable. It will be accessed as an INTEGER(4) in the body of `thread_proc` because of the `POINTER` attribute.

Example

```
INTERFACE
 integer(4) FUNCTION thread_proc(arg)
!DEC$ ATTRIBUTES STDCALL, ALIAS:"_thread_proc" :: thread_proc
 integer(4),POINTER :: arg
 END FUNCTION
END INTERFACE
```

In this second interface example, the argument is passed as an address of some undefined data in memory. You must create a pointer to a specific data type variable with the POINTER attribute and assign the `lpThreadParameter` argument to it in the body of the thread_proc routine. You can pass a pointer to an entire block of data with this method.

Example

```
INTERFACE
  integer(4) FUNCTION thread_proc(lpThreadParameter)
  !DEC$ ATTRIBUTES STDCALL, ALIAS:"_thread_proc" :: thread_proc
  integer(INT_PTR_KIND()) lpThreadParameter
 END FUNCTION
END INTERFACE
```

Example 1

```
Program TESTPROC0
   use ifcore
   use ifmt
   INTERFACE
    integer(4) FUNCTION Thread_Proc0(arg)
     !DEC$ ATTRIBUTES STDCALL, ALIAS:"_thread_proc0" :: Thread_Proc0
     integer(4),POINTER :: arg
    END FUNCTION
   END INTERFACE
   integer(INT_PTR_KIND()) ThreadHandle
   integer(INT_PTR_KIND()), PARAMETER :: security = 0
   integer(INT_PTR_KIND()), PARAMETER :: stack_size = 0
   integer(INT_PTR_KIND()) :: thread_id
   integer(4) :: ivalue0 = 12345678
   ThreadHandle = CreateThread(security,stack_size,Thread_Proc0,loc(ivalue0), &
      CREATE_SUSPENDED, thread_id )
```

Example 1

```
   iretlog = SetThreadPriority(ThreadHandle, THREAD_PRIORITY_BELOW_NORMAL )
   iretint = ResumeThread(ThreadHandle)
   call sleepqq(100) ! let io complete
end

integer(4) function Thread_Proc0(arg)
   USE IFCORE
   USE IFMT
   !DEC$ ATTRIBUTES STDCALL, ALIAS:"_thread_proc0" :: Thread_Proc0
   integer(4), POINTER :: arg
   write(6,*) "The value of the Thread_Proc0 argument is ",arg
   Thread_Proc0 = 0
   call ExitThread(0)
end function
```

The resulting output will be similar to the following example:

```
The value of the Thread_Proc0 argument is 12345678
```

Example 2

```
Program TESTPROC1
   use ifcore
   use ifmt
   INTERFACE
     integer(4) FUNCTION Thread_Proc1(lpThreadParameter)
     !DEC$ ATTRIBUTES STDCALL,ALIAS:"_thread_proc1" :: Thread_Proc1
     integer(INT_PTR_KIND()) lpThreadParameter
     END FUNCTION
   END INTERFACE
   integer(INT_PTR_KIND()) ThreadHandle1
   integer(INT_PTR_KIND()), PARAMETER :: security = 0
   integer(INT_PTR_KIND()), PARAMETER :: stack_size = 0
   integer(INT_PTR_KIND()) :: thread_id
   integer(4) :: ivalue1(5) = (/1,2,3,4,5/)
   ThreadHandle1 = CreateThread(security,stack_size,Thread_Proc1,loc(ivalue1(1)), &
      CREATE_SUSPENDED, thread_id )
   iretlog = SetThreadPriority(ThreadHandle1, THREAD_PRIORITY_BELOW_NORMAL )
   iretint = ResumeThread(ThreadHandle1)
   call sleepqq(100) ! let IO complete
end
```

Example 2

```
integer(4) function Thread_Proc1(lpThreadParameter)
  USE IFCORE
  USE IFMT
  !DEC$ ATTRIBUTES STDCALL, ALIAS:"_thread_proc1" :: Thread_Proc1
  integer(INT_PTR_KIND()) lpThreadParameter
  integer(4) arg(5)
  POINTER(parg,arg)
  parg = lpThreadParameter
  write(6,*) "The value of the Thread_Proc1 argument is ",arg
  Thread_Proc1 = 0
  call ExitThread(0)
end function
```

The resulting output will be similar to the following example:

```
The value of the Thread_Proc1 argument is 1 2 3 4 5
```

Sharing Resources

Each thread has its own stack and its own copy of the CPU registers. Other resources, such as files, units, static data, and heap memory, are shared by all threads in the process. Threads using these common resources must coordinate their work. There are several ways to synchronize resources:

- Critical section - A block of code that accesses a non-shareable resource. Critical sections are typically used to restrict access to data or code that can only be used by one thread at a time within a process (for example, modification of shared data in a common block).
- MUTual EXclusion object (Mutex) - A mechanism that allows only one thread at a time to access a resource. Mutexes are typically used to restrict access to a system resource that can only be used by one thread at a time (for example, a printer), or when sharing might produce unpredictable results.
- Semaphore - A counter that regulates the number of threads that can use a resource. Semaphores are typically used to control access to a specified number of identical resources.
- Event - An event object announces that an event has happened to one or more threads.

The state of each of these objects is either signaled or not-signaled. A signaled state indicates a resource is available for a process or thread to use it. A not-signaled state indicates the resource is in use.

The routines described in the following sections manage the creation, initialization, and termination of resource sharing mechanisms. Some of them change the state to signaled from not-signaled. The routines `WaitForSingleObject` and `WaitForMultipleObjects` also change the signal status of an object. For information on these functions, see Synchronizing Threads.

This section also contains information about:

- Memory Use and Thread Stacks
- I/O Operations

Critical Sections

Before you can synchronize threads with a critical section, you must initialize it by calling `InitializeCriticalSection`. Call `EnterCriticalSection` when beginning to process the global variable, and `LeaveCriticalSection` when the application is finished with it. Both `EnterCriticalSection` and `LeaveCriticalSection` can be called several times within an application.

Mutexes

`CreateMutex` creates a mutex object. It returns an error if the mutex already exists (one by the same name was created by another process or thread). Call `GetLastError` after calling `CreateMutex` to look for the error status `ERROR_ALREADY_EXISTS`. You can also use the `OpenMutex` function to determine whether or not a named mutex object exists. When called, `OpenMutex` returns the object's handle if it exists, or null if a mutex with the specified name is not found. Using `OpenMutex` does not change a mutex object to a signaled state; this is accomplished by one of the wait routines described in Synchronizing Threads.

`ReleaseMutex` changes a mutex from the not-signaled state to the signaled state. This function only has an effect if the thread calling it also owns the mutex. When the mutex is in a signaled state, any thread waiting for it can acquire it and begin executing.

Semaphores

Functions for handling semaphores are nearly identical to functions that manage mutexes. `CreateSemaphore` creates a semaphore, specifying an initial as well as a maximum count for the number of threads that can access the resource. `OpenSemaphore`, like `OpenMutex`, returns the handle of the named semaphore object, if it exists. The handle can then be used in any function that requires it (such as one of the wait functions described in Synchronizing Threads). Calling `OpenSemaphore` does not reduce a resource's available count; this is accomplished by the function waiting for the resource.

Use `ReleaseSemaphore` to increase the available count for a resource by a specified amount. You can call this function when the thread is finished with the resource. Another possible use is to call `CreateSemaphore`, specifying an initial count of zero to protect the resource from access during an initialization process. When the application has finished its initialization, call `ReleaseSemaphore` to increase the resource's count to its maximum.

Events

Event objects can trigger execution of other threads. You can use events if one thread provides data to several other threads. An event object is created by the `CreateEvent` function. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event. A manual-reset event is one whose state remains signaled until it is explicitly reset by a call to `ResetEvent`. An auto-reset event is automatically reset by the system when a single waiting thread is released.

Use either `SetEvent` or `PulseEvent` to set an event object's state to signaled. `OpenEvent` returns a handle to the event, which can be used in other function calls. `ReleaseEvent` releases ownership of the event.

Memory Use and Thread Stacks

Because each thread has its own stack, you can avoid potential collisions over data items by using as little static data as possible. Design your program to use automatic stack variables for all data that can be private to a thread. All the variables declared in a multithread routine are by default static and shared among the threads. If you do not want one thread to overwrite a variable used by another, you can do one of the following:

- Declare the variable as `AUTOMATIC`.
- Create a vector of variable values, one for each thread, so that the variable values for different threads are in different storage locations. (You can use the single integer parameter passed by `CreateThread` as an index to identify the thread.)
- Use Thread Local Storage (TLS).

Variables declared as automatic are placed on the stack, which is part of the thread context saved with the thread. Automatic variables within procedures are discarded when the procedure completes execution.

I/O Operations

Although files and units are shared between threads, you may not need to coordinate the use of these shared resources by threads. Fortran treats each input/output statement as an atomic operation. If two separate threads try to write to the same unit and one thread's output operation has started, the operation will complete before the other thread's output operation can begin.

The operating system does not impose an ordering on threads' access to units or files. For example, the non-determinate nature of multithread applications can cause records in a sequential file to be written in a different order on each execution of the application as each

thread writes to the file. Direct access files might be a better choice than sequential files in such a case. If you cannot use direct access files, use mutexes to impose an ordering constraint on input or output of sequential files.

Certain restrictions apply to blocking functions for input procedures in QuickWin programs. For details on these restrictions, see Using QuickWin in Building Applications.

Thread Local Storage

Thread Local Storage (TLS) calls allow you to store per-thread data. TLS is the method by which each thread in a multithreaded process can allocate locations in which to store thread-specific data.

Dynamically bound (run-time) thread-specific data is supported by routines such as `TlsAlloc` (allocates an index to store data), `TlsGetValue` (retrieves values from an index), `TlsSetValue` (stores values into an index), and `TlsFree` (frees the dynamic storage). Threads allocate dynamic storage and use `TlsSetValue` to associate the index with a pointer to that storage. When a thread needs to access the storage, it calls `TlsGetValue`, specifying the index.

When all threads have finished using the index, `TlsFree` frees the dynamic storage.

Synchronizing Threads

The routines `WaitForSingleObject` and `WaitForMultipleObjects` enable threads to wait for a variety of different occurrences, such as thread completion or signals from other threads. They enable threads and processes to wait efficiently, consuming no CPU resources, either indefinitely or until a specified timeout interval has elapsed.

WaitForSingleObject takes an object handle as the first parameter and does not return until the object referenced by the handle either reaches a signaled state or until a specified timeout value elapses. The syntax is:

`WaitResult` `= WaitForSingleObject (`*`ObjectHandle`*`, [ `*`Timeout`*` ] )`

If you are using a timeout, specify the value in milliseconds as the second parameter. The value `WAIT_INFINITE` represents an infinite timeout, in which case the function waits until *`ObjectHandle`* completes.

`WaitForMultipleObjects` is similar, except that its second parameter is an array of Windows object handles. Specify the number of handles to wait for in the first parameter. This can be less than the total number of threads created, and its maximum is 64. The function can either wait until all events have completed, or resume as soon as any one of the objects completes.

Deadlocks occur when a thread waits for objects that never become available. Use the timeout parameter when there is a chance that the thread you are waiting for may never terminate.

Suspending and Resuming Threads

You can use `SuspendThread` to stop a thread from executing. `SuspendThread` is not particularly useful for synchronization because it does not control the point in the code at which the thread's execution is suspended. However, you could suspend a thread if you need to confirm the user input that would terminate the work of the thread. If confirmed, the thread is terminated; otherwise, it resumes.

If a thread is created in a suspended state, it does not begin to run until `ResumeThread` is called with a handle to the suspended thread. This can be useful for initializing the thread's state before it begins to run. Suspending a thread at creation can be useful for one-time synchronization, because `ResumeThread` ensures that the suspended thread will resume running at the starting point of its code.

Handling Errors in Multithread Programs

Use the `GetLastError` function to obtain error information if any of the multithreading routines returns an error code. Remember that it returns the error code of the last error, not necessarily the error status of the last call.

Error codes are 32-bit values. Bit 29 is reserved for application-defined error codes. You can set this bit and use `SetLastError` if you are creating your own dynamic-link library, to emulate Windows API behavior. Windows functions only call `SetLastError` when they fail, not when they succeed.

The last error code value is kept in Thread Local Storage, so that multiple threads do not overwrite each other's values.

Table of Multithread Routines

The following table lists routines available for multithread programs. For more information on these routines, see the Microsoft Developer Network (MSDN) or platform SDK documentation.

Routine	Description
`CloseHandle`	Closes an open object handle.
`CreateEvent`	Creates a named or unnamed event object.
`CreateMutex`	Creates a named or unnamed mutex object.
`CreateProcess`	Creates a new process and its primary thread.
`CreateSemaphore`	Creates a named or unnamed semaphore object.

Routine	Description
CreateThread	Creates a thread to execute within the address space of the calling process.
DeleteCriticalSection	Releases all resources used by an unowned critical section object.
DuplicateHandle	Duplicates an object handle.
EnterCriticalSection	Waits for ownership of the specified critical section object.
ExitProcess	Ends a process and all its threads.
ExitThread	Ends a thread.
GetCurrentProcess	Returns a pseudohandle for the current process.
GetCurrentProcessId	Returns the process identifier of the calling process.
GetCurrentThread	Returns a pseudohandle for the current thread.
GetCurrentThreadId	Returns the thread identifier of the calling thread.
GetExitCodeProcess	Retrieves the termination status of the specified process.
GetExitCodeThread	Retrieves the termination status of the specified thread.
GetLastError	Returns the calling thread's last-error code value.
GetPriorityClass	Returns the priority class for the specified process.
GetThreadPriority	Returns the priority value for the specified thread.

Routine	Description
InitializeCriticalSection	Initializes a critical section object.
LeaveCriticalSection	Releases ownership of the specified critical section object.
OpenEvent	Returns a handle of an existing named event object.
OpenMutex	Returns a handle of an existing named mutex object.
OpenProcess	Returns a handle of an existing process object.
OpenSemaphore	Returns a handle of an existing named semaphore object.
PulseEvent	As a single operation, sets (to signaled) and then resets the state of the specified event object after releasing the appropriate number of waiting threads.
ReleaseMutex	Releases ownership of the specified mutex object.
ReleaseSemaphore	Increases the count of the specified semaphore object by a specified amount.
ResetEvent	Sets the state of the specified event object to nonsignaled.
ResumeThread	Decrements a thread's suspend count. When the suspend count is zero, execution of the thread resumes.
SetEvent	Sets the state of the specified event object to signaled.
SetLastError	Sets the last-error code for the calling thread.
SetPriorityClass	Sets the priority class for the specified process.

Routine	Description
SetThreadPriority	Sets the priority value for the specified thread.
SuspendThread	Suspends the specified thread.
TerminateProcess	Terminates the specified process and all of its threads.
TerminateThread	Terminates a thread.
TlsAlloc	Allocates a thread local storage (TLS) index.
TlsFree	Releases a thread local storage (TLS) index, making it available for reuse.
TlsGetValue	Retrieves the value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
TlsSetValue	Stores a value in the calling thread's thread local storage (TLS) slot for a specified TLS index.
WaitForMultipleObjects	Returns either any one or all of the specified objects are in the signaled state or when the time-out interval elapses.
WaitForSingleObject	Returns when the specified object is in the signaled state or the time-out interval elapses.

If a function mentioned in this section is not listed in the preceding table, it is only available through the **USE IFWIN** statement.

Working with Multiple Processes

The multithread libraries provide a number of routines for working with multiple processes. An application can use multiple processes for functions that require a private address space and private resources, to protect them from the activities of other threads. It is usually more efficient to implement multitasking by creating several threads in one process, rather than by creating multiple processes, for these reasons:

- The system can create and execute threads more quickly than it can create processes, since the code for threads has already been mapped into the address space of the process, while the code for a new process must be loaded.
- All threads of a process share the same address space and can access the process's global variables, which can simplify communications between threads.
- All threads of a process can use open handles to resources such as files and pipes.

If you want to create an independent process that runs concurrently with the current one, use `CreateProcess`. `CreateProcess` returns a process identifier that is valid until the process terminates. `ExitProcess` stops the process and notifies all DLLs the process is terminating.

Different processes can share mutexes, events, and semaphores (but not critical sections). Processes can also optionally inherit handles from the process that created them (see online help for `CreateProcess`).

You can obtain information about the current process by calling `GetCurrentProcess` (returns a pseudohandle to its own process), and `GetCurrentProcessId` (returns the process identifier). The value returned by these functions can be used in calls to communicate with other processes. `GetExitCodeProcess` returns the exit code of a process, or an indication that it is still running.

The `OpenProcess` function opens a handle to a process specified by its process identifier. `OpenProcess` allows you to specify the handle's access rights and inheritability.

A process terminates whenever one of the following occurs:

- Any thread of the process calls `ExitProcess`
- The primary thread of the process returns
- The last thread of the process terminates
- `TerminateProcess` is called with a handle to the process

`ExitProcess` is the preferred way to terminate a process because it notifies all attached DLLs of the termination, and ensures that all threads of the process terminate. DLLs are not notified after a call to `TerminateProcess`.

Using Interprocedural Optimization (IPO)

30

Interprocedural Optimization (IPO) Overview

Interprocedural Optimization (IPO) allows the compiler to analyze your code to determine where you can benefit from specific optimizations. In many cases, the optimizations that can be applied are related to the specific architectures.

The compiler might apply the following optimizations for the listed architectures:

Architecture	Optimization
IA-32, Intel® 64, and IA-64 architectures	• inlining • constant propagation • mod/ref analysis • alias analysis • forward substitution • routine key-attribute propagation • address-taken analysis • partial dead call elimination • symbol table data promotion • common block variable coalescing • dead function elimination • unreferenced variable removal • whole program analysis • array dimension padding • common block splitting • stack frame alignment • structure splitting and field reordering • formal parameter alignment analysis • indirect call conversion • specialization

Architecture	Optimization
IA-32 and Intel® 64 architectures	• Passing arguments in registers to optimize calls and register usage
IA-64 architecture only	• removing redundant EXTEND instructions • short data section allocation • prefetch analysis

IPO is an automatic, multi-step process: compilation and linking; however, IPO supports two compilation models: single-file compilation and multi-file compilation.

Single-file compilation, which uses the `-ip` (Linux* OS and Mac OS* X) or `/Qip` (Windows* OS) option, results in one, real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

The compiler performs some single-file interprocedural optimization at the default optimization level: `-O2` (Linux* and Mac OS* X) or `/O2` (Windows*); additionally some the compiler performs some inlining for the `-O1` (Linux* and Mac OS* X) or `/O1` (Windows*) optimization level, like inlining functions marked with inlining directives.

Multi-file compilation, which uses the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) option, results in one or more mock object files rather than normal object files. (See the *Compilation* section below for information about mock object files.) Additionally, the compiler collects information from the individual source files that make up the program. Using this information, the compiler performs optimizations across functions and procedures in different source files. Inlining is the most powerful optimization supported by IPO. See Inline Function Expansion.

NOTE. Inlining and other optimizations are improved by profile information. For a description of how to use IPO with profile information for further optimization, see Profile an Application.

Mac OS* X: Intel®-based systems running Mac OS X do not support a multiple object compilation model.

Compilation

As each source file is compiled with IPO, the compiler stores an intermediate representation (IR) of the source code in a mock object file, which includes summary information used for optimization. The mock object files contain the IR, instead of the normal object code. Mock object files can be ten times larger, and in some cases more, than the size of normal object files.

During the IPO compilation phase only the mock object files are visible. The Intel compiler does not expose the real object files during IPO unless you also specify the `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows) option.

Linkage

When you link with the `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows) option the compiler is invoked a final time. The compiler performs IPO across all object files that have an IR equivalent. The mock objects must be linked with the Intel compiler or by using the Intel linking tools. The compiler calls the linkers indirectly by using aliases (or wrappers) for the native linkers, so you must modify make files to account for the different linking tool names. For information on using the linking tools, see Using IPO; see the Linking Tools and Options topic for detailed information.

CAUTION. Linking the mock object files with ld (Linux and Mac OS X) or link.exe (Windows) will cause linkage errors. You must use the Intel linking tools to link mock object files.

During the compilation process, the compiler first analyzes the summary information and then produces mock object files for source files of the application.

Whole program analysis

The compiler supports a large number of IPO optimizations that either can be applied or have the effectiveness greatly increased when the whole program condition is satisfied.

Whole program analysis, when it can be done, enables many interprocedural optimizations. During the analysis process, the compiler reads all Intermediate Representation (IR) in the mock file, object files, and library files to determine if all references are resolved and whether or not a given symbol is defined in a mock object file. Symbols that are included in the IR in a mock object file for both data and functions are candidates for manipulation based on the results of whole program analysis.

There are two types of whole program analysis: object reader method and table method. Most optimizations can be applied if either type of whole program analysis determine that the whole program conditions exists; however, some optimizations require the results of the object reader method, and some optimizations require the results of table method.

NOTE. The IPO report provides details about whether whole program analysis was satisfied and indicate the method used during IPO compilation.

In the first type of whole program analysis, the object reader method, the object reader emulates the behavior of the native linker and attempts to resolve the symbols in the application. If all symbols are resolved correctly, the whole program condition is satisfied. This type of whole program analysis is more likely to detect the whole program condition.

Often the object files and libraries accessed by the compiler do not represent the whole program; there are many dependencies to well-known libraries. IPO linking, whole program analysis, determines whether or not the whole program can be detected using the available compiler resources.

The second type of whole program analysis, the table method, is where the compiler analyzes the mock object files and generates a call-graph.

The compiler contains detailed tables about all of the functions for all important language-specific libraries, like the Fortran runtime libraries. In this second method, the compiler constructs a call-graph for the application. The compiler then compares the function table and application call-graph. For each unresolved function in the call-graph, the compiler attempts to resolve the calls. If the compiler can resolve the functions call, the whole program condition exists.

Interprocedural Optimization (IPO) Quick Reference

IPO is a two step process: compile and link. See Using IPO.

Linux* and Mac OS* X	Windows*	Description
`-ipo` or `-ipo`*N*	`/Qipo` or `/Qipo`*N*	Enables interprocedural optimization for multi-file compilations. Normally, multi-file compilations result in a single object file only. Passing an integer value for *N* allows you to specify number of true

Linux* and Mac OS* X	Windows*	Description
		object files to generate; the default value is 0, which means the compiler determines the appropriate number of object files to generate. (See IPO for Large Programs.)
`-ipo-separate`	`/Qipo-separate`	Instructs the compiler to generate a separate, real object file for each mock object file. Using this option overrides any integer value passed for `ipo`*N*. (See IPO for Large Programs for specifics.)
`-ip`	`/Qip`	Enables interprocedural optimizations for single file compilations. Instructs the compiler to generate a separate, real object file for each source file.

Additionally, the compiler supports options that provide support for compiler-directed or developer-directed inline function expansion.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Using IPO

This topic discusses how to use IPO from a command line. For specific information on using IPO from within an Integrated Development Environment (IDE), refer to the appropriate section in Building Applications.

Compiling and Linking Using IPO

The steps to enable IPO for compilations targeted for IA-32, Intel® 64, and IA-64 architectures are the same: compile and link.

First, compile your source files with `-ipo` (Linux* and Mac OS* X) or `/Qipo` (Windows*) as demonstrated below:

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo -c a.f90 b.f90 c.f90`
Windows*	`ifort /Qipo /c a.f90 b.f90 c.f90`

The output of the above example command differs according to operating system:

- Linux and Mac OS X: The commands produce `a.o`, `b.o`, and `c.o` object files.
- Windows: The commands produce `a.obj`, `b.obj`, and `c.obj` object files.

Use `-c` (Linux and Mac OS X) or `/c` (Windows) to stop compilation after generating `.o` or `.obj` files. The output files contain compiler intermediate representation (IR) corresponding to the compiled source files. (See the section below on capturing the intermediate IPO output.)

Second, link the resulting files. The following example command will produce an executable named `app`:

Operating System	Example Command
Linux and Mac OS X	`ifort -o app a.o b.o c.o`
Windows	`ifort /exe:app a.obj b.obj c.obj`

The command invoke the compiler on the objects containing IR and creates a new list of objects to be linked. Alternately, you can use the `xild` (Linux and Mac OS X) or `xilink` (Windows) tool, with the appropriate linking options.

Combining the Steps

The separate compile and link commands demonstrated above can be combined into a single command, as shown in the following examples:

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo -o app a.f90 b.f90 c.f90`
Windows	`ifort /Qipo /exe:app a.f90 b.f90 c.f90`

The `ifort` command, shown in the examples above, calls `GCC ld` (Linux and Mac OS X) or Microsoft* `link.exe` (Windows only) to link the specified object files and produce the executable application, which is specified by the `-o` (Linux and Mac OS X) or `/exe` (Windows) option.

Capturing Intermediate IPO Output

The `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows*) and `-ipo-S` (Linux and Mac OS X) or `/Qipo-S` (Windows) options are useful for analyzing the effects of multi-file IPO, or when experimenting with multi-file IPO between modules that do not make up a complete program.

- Use the `-ipo-c` option to optimize across files and produce an object file. The option performs optimizations as described for the `-ipo` option but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.s` (Linux and Mac OS X) or `ipo_out.obj` (Windows).
- Use the `-ipo-S` option to optimize across files and produce an assembly file. The option performs optimizations as described for `-ipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.s` (Linux) or `ipo_out.asm` (Windows).

For both options, you can use the `-o` (Linux and Mac OS X) or `/exe` (Windows) option to specify a different name.

These options generate multiple outputs if multi-object IPO is being used. The name of the first file is taken from the value of the `-o` (Linux and Mac OS X) or /exe (Windows) option.

The names of subsequent files are derived from the first file with an appended numeric value to the file name. For example, if the first object file is named `foo.o` (Linux and Mac OS X) or `foo.obj` (Windows), the second object file will be named `foo1.o` or `foo1.obj`.

You can use the object file generated with the `-ipo-c` (Linux and Mac OS X) or `/Qipo-c` (Windows) option, but you will not get the same benefits from the optimizations applied that would otherwise if the whole program analysis condition had been satisfied.

The object file created using the `-ipo-c` option is a real object file, in contrast to the mock file normally generated using IPO; however, the generated object file is significantly different than the mock object file. It does not contain the IR information needed to fully optimize the application using IPO.

The compiler generates a message indicating the name of each object or assembly file it generates. These files can be added to the real link step to build the final application.

IPO-Related Performance Issues

There are some general optimization guidelines for IPO that you should keep in mind:

- Large IPO compilations might trigger internal limits of other compiler optimization phases.
- Combining IPO and PGO can be a key technique for C++ applications. The `-O3`, `-ipo`, and `-prof-use` (Linux* and Mac OS* X) or `/O3`, `/Qipo`, `/Qprof-use` (Windows*) options can result in significant performance gains.
- IPO benefits C more than C++, since C++ compilations include intra-file inlining by default.
- Applications where the compiler does not have sufficient intermediate representation (IR) coverage to do whole program analysis might not perform as well as those where IR information is complete.

In addition to the general guidelines, there are some practices to avoid while using IPO. The following list summarizes the activities to avoid:

- Do not use the link phase of an IPO compilation using mock object files produced for your application by a different compiler. The Intel® Compiler cannot inspect mock object files generated by other compilers for optimization opportunities.
- Do not link mock files with the `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*) option unless the mock files were also compiled with the `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) option.
- Update make files to call the appropriate Intel linkers when using IPO from scripts. For Linux and Mac OS X, replaces all instances of `ld` with `xild`; for Windows, replace all instances of `link` with `xilink`.
- Update make file to call the appropriate Intel archiver. Replace all instances of `ar` with `xiar`.

IPO for Large Programs

In most cases, IPO generates a single object file for the link-time compilation. This behavior is not optimal for very large programs, perhaps even making it impossible to use `-ipo` (Linux* and Mac OS* X) or `/Qipo` (Windows*) on the application.

The compiler provides two methods to avoid this problem. The first method is an automatic size-based heuristic, which causes the compiler to generate multiple object files for large link-time compilations. The second method is to manually instruct the compiler to perform multi-object IPO.

- Use the `-ipo`*N* (Linux and Mac OS X) or `/Qipo`*N* (Windows) option and pass an integer value in the place of *N*.

- Use the `-ipo-separate` (Linux and Mac OS X) or `/Qipo-separate` (Windows) option.

  The number of true object files generated by the link-time compilation is invisible to the user unless either the `-ipo-c` or `-ipo-S` (Linux and Mac OS X) or `/Qipo-c` or `/Qipo-S` (Windows) option is used.

Regardless of the method used, it is best to use the compiler defaults first and examine the results. If the defaults do not provide the expected results then experiment with generating more object files.

You can use the `-ipo-jobs` (Linux and Mac OS X) or `/Qipo-jobs` (Windows) option to control the number of commands, or jobs, executed during parallel builds.

Using -ipo*N* or /Qipo*N* to Create Multiple Object Files

If you specify `-ipo0` (Linux and Mac OS X) or `/Qipo0` (Windows), which is the same as not specifying a value, the compiler uses heuristics to determine whether to create one or more object files based on the expected size of the application. The compiler generates one object file for small applications, and two or more object files for large applications. If you specify any value greater than 0, the compiler generates that number of object files, unless the value you pass a value that exceeds the number of source files. In that case, the compiler creates one object file for each source file then stops generating object files.

The following example commands demonstrate how to use `-ipo2` (Linux and Mac OS X) or `/Qipo2` (Windows) to compile large programs.

Operating System	Example Command
Linux and Mac OS X	`ifort -ipo2 -c a.f90 b.f90`
Windows	`ifort /Qipo2 /c a.f90 b.f90`

Because the example command shown above, the compiler generates object files using an OS-dependent naming convention. On Linux and Mac OS X, the example command results in object files named `ipo_out.o`, `ipo_out1.o`, `ipo_out2.o`, and `ipo_out3.o`. On Windows, the file names follow the same convention; however, the file extensions will be `.obj`.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

Creating the Maximum Number of Object Files

Using `-ipo-separate` (Linux and Mac OS X) or `/Qipo-separate` (Windows) allows you to force the compiler to generate the maximum number of true object files that the compiler will support during multiple object compilation.

For example, if you passed example commands similar to the following:

Operating System	Example Command
Linux and Mac OS X	`ifort a.o b.o c.o -ipo-separate -ipo-c`
Windows	`ifort a.obj b.obj c.obj /Qipo-separate /Qipo-c`

The compiler will generate multiple object file, which use the same naming convention discussed above. The first object file contains global variables. The other object files contain code for the functions or routines used the source files.

Link the resulting object files as shown in Using IPO or Linking Tools and Options.

Considerations for Large Program Compilation

For many large programs, compiling with IPO can result in a single, large object file. Compiling to produce large objects can create problems for efficient compilation. During compilation, the compiler attempts to swap the memory usage during compiles; a large object file might result in poor swap usage, which could result in out-of-memory message or long compilation times. Using multiple, relatively small object files during compilation causes the system to use resources more efficiently.

Understanding Code Layout and Multi-Object IPO

One of the optimizations performed during an IPO compilation is code layout. The analysis performed by the compiler during multi-file IPO determines a layout order for all of the routines for which it has intermediate representation (IR) information. For a multi-object IPO compilation, the compiler must tell the linker about the desired order.

If you are generating an executable in the link step, the compiler does all of this automatically. However, if you are generating object files instead of an executable, the compiler generates a layout script, which contains the correct information needed to optimally link the executable when you are ready to create it.

This linking tool script must be taken into account if you use either `-ipo-c` or `-ipo-S` (Linux*) or `/Qipo-c` or `/Qipo-S` (Windows*). With these options, the IPO compilation and actual linking are done by different invocations of the compiler. When this occurs, the compiler issues a message indicating that it is generating an explicit linker script, `ipo_layout.script`.

The Windows linker (link.exe) automatically collates these sections lexigraphically in the desired order.

The compiler first puts each routine in a named text section that varies depending on the operating system:

Windows:

- The first routine is placed in `.text$00001`, the second is placed in `.text$00002`, and so on.

Linux:

- The first routine is placed in `.text00001`, the second is placed in `.text00002`, and so on. It then generates a linker script that tells the linker to first link contributions from `.text00001`, then `.text00002`.

When `ipo_layout.script` is generated, you should modify your link command if you want to use the script to optimize code layout:

Example

```
--script=ipo_layout.script
```

If your application already requires a custom linker script, you can place the necessary contents of `ipo_layout.script` in your script.

The layout-specific content of `ipo_layout.script` is at the beginning of the description of the `.text` section. For example, to describe the layout order for 12 routines:

Example output

```
.text     :
{
*(.text00001) *(.text00002) *(.text00003) *(.text00004) *(.text00005)
*(.text00006) *(.text00007) *(.text00008) *(.text00009) *(.text00010)
*(.text00011) *(.text00012)
...
```

For applications that already require a linker script, you can add this section of the `.text` section description to the customized linker script. If you add these lines to your linker script, it is desirable to add additional entries to account for future development. The addition is harmless, since the "r;*("¦)" syntax makes these contributions optional.

If you choose to not use the linker script your application will still build, but the layout order will be random. This may have an adverse affect on application performance, particularly for large applications.

Creating a Library from IPO Objects

Linux* and Mac OS* X

Libraries are often created using a library manager such as `lib`. Given a list of objects, the library manager will insert the objects into a named library to be used in subsequent link steps.

Example
`xiar cru user.a a.o b.o`

The above command creates a library named `user.a` containing the `a.o` and `b.o` objects.

If the objects have been created using `-ipo-c` then the archive will not only contain a valid object, but the archive will also contain intermediate representation (IR) for that object file. For example, the following example will produce `a.o` and `b.o` that may be archived to produce a library containing both object code and IR for each source file.

Example
`ifort -ipo -c a.f90 b.f90`

This program will invoke the compiler on the IR saved in the object file and generate a valid object that can be inserted into a library.

Using `xiar` is the same as specifying `xild -lib`.

Mac OS X Only

When using `xilibtool`, specify `-static` to generate static libraries, or specify `-dynamiclib` to create dynamic libraries. For example, the following example command will create a static library named `mylib.a` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -static -o mylib.a a.o b.o c.o
```

Alternately, the following example command will create a dynamic library named `mylib.dylib` that includes the `a.o`, `b.o`, and `c.o` objects.

Example

```
xilibtool -dynamic -o mylib.dylib a.o b.o c.o
```

Specifying `xilibtool` is the same as specifying `xild -libtool`.

Windows* Only

Create libraries using `xilib` or `xilink -lib` to create libraries of IPO mock object files and link them on the command line.

For example, assume that you create three mock object files by using a command similar to the following:

Example

```
ifort /c /Qipo a.obj b.obj c.obj
```

Further assume `a.obj` contains the main subprogram. You can enter commands similar to the following to link the objects.

Example

```
xilib -out:main.lib b.obj c.obj

or

xilink -lib -out:main.lib b.obj c.obj
```

You can link the library and the main program object file by entering a command similar to the following:

Example
`xilink -out:result.exe a.obj main.lib`

Requesting Compiler Reports with the xi* Tools

The compiler option `-opt-report` (Linux* and Mac OS* X) or `/Qopt-report` (Windows*) generates optimization reports with different levels of detail. Related compiler options described in Compiler Reports Quick Reference allow you to specify the phase, direct output to a file (instead of stderr), and request reports from all routines with names containing a string as part of their name.

The xi* tools are used with interprocedural optimization (IPO) during the final stage of IPO compilation. You can request compiler reports to be generated during the final IPO compilation by using certain options. The supported xi* tools are:

- Linker tools: `xilink` (Windows) and `xild` (Linux and Mac OS X)
- Library tools: `xilib` (Windows), `xiar` (Linux and Mac OS X), `xilibtool` (Mac OS X)

The following tables lists the compiler report options for the xi* tools. These options are equivalent to the corresponding compiler options, but occur during the final IPO compilation.

Tool Option	Description
`-qopt-report`[=*n*]	Enables optimization report generation with different levels of detail, directed to stderr. Valid values for *n* are 0 through 3. By default, when you specify this option without passing a value the compiler will generate a report with a medium level of detail.
`-qopt-report-file=`*file*	Generates an optimization report and directs the report output to the specified *file* name. If you omit the path, the file is created in the current directory. To create the file in a different directory, specify the full path to the output file and its file name.
`-qopt-report-phase=`*name*	Specifies the optimization phase *name* to use when generating reports. If you do not specify a phase the compiler defaults to all. You can request a list of all available phase by using the -qopt-report-help option.

Tool Option	Description
`-qopt-report-routine=`*`string`*	Generates reports from all routines with names containing a *`string`* as part of their name. If not specified, the compiler will generate reports on all routines.
`-qopt-report-help`	Displays the optimization phases available.

To understand the compiler reports, use the links provided in Compiler Reports Overview.

Inline Expansion of Functions

Inline Function Expansion

Because inline function expansion does not require that the applications meet the criteria for whole program analysis normally require by IPO, this optimization is one of the primary optimizations used in Interprocedural Optimization (IPO). For function calls that the compiler believes are frequently executed, the Intel® compiler often decides to replace the instructions of the call with code for the function itself.

In the compiler, inline function expansion typically favors relatively small user functions over functions that are relatively large. This optimization improves application performance by performing the following:

- Removing the need to set up parameters for a function call
- Eliminating the function call branch
- Propagating constants

Function inlining can improve execution time by removing the runtime overhead of function calls; however, function inlining can increase code size, code complexity, and compile times. In general, when you instruct the compiler to perform function inlining, the compiler can examine the source code in a much larger context, and the compiler can find more opportunities to apply optimizations.

Specifying `-ip` (Linux* and Mac OS* X) or `/Qip` (Windows*), single-file IP, causes the compiler to perform inline function expansion for calls to procedures defined within the current source file; in contrast, specifying `-ipo` (Linux and Mac OS X) or `/Qipo` (Windows), multi-file IPO, causes the compiler to perform inline function expansion for calls to procedures defined in other files.

CAUTION. Using the `-ip` and `-ipo` (Linux and Mac OS X) or `/Qip` and `/Qipo` (Windows) options can in some cases significantly increase compile time and code size.

Selecting Routines for Inlining

The compiler attempts to select the routines whose inline expansions will provide the greatest benefit to program performance. The selection is done using the default heuristics. The inlining heuristics used by the compiler differ based on whether or not you use Profile-Guided Optimizations (PGO): `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

When you use PGO with `-ip` or `-ipo` (Linux and Mac OS X) or `/Qip` or `/Qipo` (Windows), the compiler uses the following guidelines for applying heuristics:

- The default heuristic focuses on the most frequently executed call sites, based on the profile information gathered for the program.
- The default heuristic always inlines very small functions that meet the minimum inline criteria.

PGO (Windows)

Combining IPO and PGO produces better results than using IPO alone. PGO produces dynamic profiling information that can usually provide better optimization opportunities than the static profiling information used in IPO.

The compiler uses characteristics of the source code to estimate which function calls are executed most frequently. It applies these estimates to the PGO-based guidelines described above. The estimation of frequency, based on static characteristics of the source, is not always accurate.

Avoid using static profile information when combining PGO and IPO; with static profile information, the compiler can only estimate the application performance for the source files being used. Using dynamically generated profile information allows the compiler to accurately determine the real performance characteristics of the application.

Compiler Directed Inline Expansion of User Functions

Without directions from the user, the compiler attempts to estimate what functions should be inlined to optimize application performance. See Criteria for Inline Function Expansion for more information.

The following options are useful in situations where an application can benefit from user function inlining but does not need specific direction about inlining limits. Except where noted, these options are supported on IA-32, Intel® 64, and IA-64 architectures.

Linux* and Mac OS* X	Windows*	Effect
`-inline-level`	`/Ob`	Specifies the level of inline function expansion. Depending on the value

Linux* and Mac OS* X	Windows*	Effect
		specified, the option can disable or enable inlining. By default, the option enables inlining of any function if the compiler believes the function can be inlined. For more information, see the following topic: • `-inline-level` compiler option
`-ip-no-inlining`	`/Qip-no-inlining`	Disables only inlining normally enabled by the following options: • Linux and Mac OS X: `-ip` or `-ipo` • Windows: `/Qip`, `/Qipo`, or `/Ob2` No other IPO optimization are disabled. For more information, see the following topic: • `-ip-no-inlining` compiler option
`-ip-no-pinlining`	`/Qip-no-pinlining`	Disables partial inlining normally enabled by the following options: • Linux and Mac OS X: `-ip` or `-ipo` • Windows: `/Qip` or `/Qipo` No other IPO optimization are disabled.

Linux* and Mac OS* X	Windows*	Effect
		For more information, see the following topic: • -ip-no-pinlining compiler option
-inline-debug-info	/Qinline-debug-info	Keeps source information for inlined functions. The additional source code can be used by the Intel® Debugger track the user-defined call stack while using inlining. To use this option you must also specify an additional option to enable debugging: • Linux: -g • Mac OS X: This option is not supported. • Windows: /debug For more information, see the following topic: • -inline-debug-info compiler option

Developer Directed Inline Expansion of User Functions

In addition to the options that support compiler directed inline expansion of user functions, the compiler also provides compiler options that allow you to more precisely direct when and if inline function expansion occurs.

The compiler measures the relative size of a routine in an abstract value of intermediate language units, which is approximately equivalent to the number of instructions that will be generated. The compiler uses the intermediate language unit estimates to classify routines and functions as relatively small, medium, or large functions. The compiler then uses the estimates to determine when to inline a function; if the minimum criteria for inlining is met and all other things are equal, the compiler has an affinity for inlining relatively small functions and not inlining relative large functions.

The following developer directed inlining options provide the ability to change the boundaries used by the inlining optimizer to distinguish between small and large functions. These options are supported on IA-32, Intel® 64, and IA-64 architectures.

In general, you should use the `-inline-factor` (Linux* and Mac OS* X) and `/Qinline-factor` (Windows*) option before using the individual inlining options listed below; this single option effectively controls several other upper-limit options.

Linux* and Mac OS* X	Windows*	Effect
`-inline-factor`	`/Qinline-factor`	Controls the multiplier applied to all inlining options that define upper limits: `inline-max-size`, `inline-max-total-size`, `inline-max-per-routine`, and `inline-max-per-compile`. While you can specify an individual increase in any of the upper-limit options, this single option provides an efficient means of controlling all of the upper-limit options with a single command. By default, this option uses a multiplier of 100, which corresponds to a factor of 1. Specifying 200 implies a factor of 2, and so on. Experiment with the multiplier carefully. You could increase the upper limits to allow too much inlining, which might result in your system running out of memory. For more information, see the following topic: • `-inline-factor` compiler option

Linux* and Mac OS* X	Windows*	Effect
-inline-forceinline	/Qinline-forceinline	Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so. Typically, the compiler targets functions that have been marked for inlining based on the presence of directives, like DEC$ ATTRIBUTES FORCEINLINE, in the source code; however, all such directives in the source code are treated only as suggestions for inlining. The option instructs the compiler to view the inlining suggestion as mandatory and inline the marked function if it can be done legally. For more information, see the following topic: • -inline-forceinline compiler option
-inline-min-size	/Qinline-min-size	Redefines the maximum size of small routines; routines that are equal to or smaller than the value specified are more likely to be inlined. For more information, see the following topic: • -inline-min-size compiler option

Linux* and Mac OS* X	Windows*	Effect
-inline-max-size	/Qinline-max-size	Redefines the minimum size of large routines; routines that are equal to or larger than the value specified are less likely to be inlined. For more information, see the following topic: • -inline-max-size compiler option
-inline-max-total-size	/Qinline-max-total-size	Limits the expanded size of inlined functions. For more information, see the following topic: • -inline-max-total-size compiler option
-inline-max-per-routine	/Qinline-max-per-routine	Limits the number of times inlining can be applied within a routine. For more information, see the following topic: • -inline-max-per-routine compiler option
-inline-max-per-compile	/Qinline-max-per-compile	Limits the number of times inlining can be applied within a compilation unit. The compilation unit limit depends on the whether or not you specify the -ipo (Linux and Mac OS X) or /Qipo (Windows) option. If you enable IPO, all source files that are part of the compilation are considered

Linux* and Mac OS* X	Windows*	Effect
		one compilation unit. For compilations not involving IPO each source file is considered an individual compilation unit. For more information, see the following topic: • `-inline-max-per-compile` compiler option

Using Profile-Guided Optimization (PGO)

31

Profile-Guided Optimizations Overview

Profile-guided Optimization (PGO) improves application performance by reorganizing code layout to reduce instruction-cache problems, shrinking code size, and reducing branch mispredictions. PGO provides information to the compiler about areas of an application that are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

PGO consists of three phases or steps.

1. Step one is to instrument the program. In this phase, the compiler creates and links an instrumented program from your source code and special code from the compiler.
2. Step two is to run the instrumented executable. Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.
3. Step three is a final compilation. When you compile a second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.

See Profile-guided Optimization Quick Reference for information about the supported options and Profile an Application for specific details about using PGO from a command line.

PGO enables the compiler to take better advantage of the processor architecture, more effective use of instruction paging and cache memory, and make better branch predictions. PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. (Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.)
- Detect and do not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Interprocedural Optimization (IPO) and PGO can affect each other; using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations. Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Performance Improvements with PGO

PGO works best for code with many frequently executed branches that are difficult to predict at compile time. An example is the code with intensive error-checking in which the error conditions are false most of the time. The infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly. Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

When you use PGO, consider the following guidelines:

- Minimize changes to your program after you execute the instrumented code and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated. (If you modify your program the compiler issues a warning that the dynamic information does not correspond to a modified function.)
- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.
- Know the sections of your code that are the most heavily used. If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.
- Different data sets can result in different algorithms being called. The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits. If it takes multiple data sets to accurately characterize application performance then execute the application with all data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Profile-Guided Optimization (PGO) Quick Reference

Profile-Guided Optimization consists of three phases (or steps):

1. Generating instrumented code by compiling with the `-prof-gen` (Linux* OS and Mac OS* X) or `/Qprof-gen` (Windows* OS) option when creating the instrumented executable.
2. Running the instrumented executable, which produces dynamic-information (.dyn) files.
3. Compiling the application using the profile information using the `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) option.

The figure illustrates the phases and the results of each phase.

See Profile an Application for details about using each phase.

The following table lists the compiler options used in PGO:

Linux* and Mac OS* X	Windows*	Effect
`-prof-gen`	`/Qprof-gen`	Instruments a program for profiling to get the execution counts of each basic block. The option is used in phase 1 (instrumenting the code) to instruct the compiler to produce instrumented code for your object files in preparation for instrumented execution. By default, each instrumented execution creates one dynamic-information (dyn) file for each executable and (on Windows OS) one for each DLL invoked by the application. You can specify keywords, such as `-prof-gen=default` (Linux and Mac OS X) or `/Qprof-gen:default` (Windows). The keywords control the amount of source information gathered during phase 2 (run the instrumented executable). The `prof-gen` keywords are: • Specify `default` (or omit the keyword) to request profiling information for use with the `prof-use` option and optimization

Linux* and Mac OS* X	Windows*	Effect
		when the instrumented application is run (phase 2). • Specify `srcpos` or `globdata` to request additional profiling information for the `code coverage` and `test prioritization` tools when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file. • Specify `globdata` to request additional profiling information for data ordering optimization when the instrumented application is run (phase 2). The phase 1 compilation creates an spi file. If you are performing a parallel make, this option will not affect it.
`-prof-use`	`/Qprof-use`	Instructs the compiler to produce a profile-optimized executable and merges available dynamic-information (dyn) files into a pgopti.dpi file. This option implicitly invokes the `profmerge` tool. The dynamic-information files are produced in phase 2 when you run the instrumented executable.

Linux* and Mac OS* X	Windows*	Effect
		If you perform multiple executions of the instrumented program to create additional dynamic-information files that are newer than the current summary pgopti.dpi file, this option merges the dynamic-information files again and overwrites the previous pgopti.dpi file (you can set the environment variable PROF_NO_CLOBBER to prevent the previous dpi file from being overwritten). When you compile with prof-use, all dynamic information and summary information files should be in the same directory (current directory or the directory specified by the `prof-dir` option). If you need to use certain `profmerge` options not available with compiler options (such as specifying multiple directories), use the `profmerge` tool. For example, you can use `profmerge` to create a new summary dpi file before you compile with the `prof-use` option to create the optimized application. You can specify keywords, such as `-prof-gen=weighted` (Linux and Mac OS X) or `/Qprof-gen:weighted` (Windows). If you omit the `weighted` keyword, the merged dynamic-information

Linux* and Mac OS* X	Windows*	Effect
		(dyn) files will be weighted proportionally to the length of time each application execution runs. If you specify the `weighted` keyword, the profiler applies an equal weighting (regardless of execution times) to the dyn file values to normalize the data counts. This keyword is useful when the execution runs have different time durations and you want them to be treated equally. When you use `prof-use`, you can also specify the `prof-file` option to rename the summary dpi file and the `prof-dir` option to specify the directory for dynamic-information (dyn) and summary (dpi) files. Linux: • Using this option with `-prof-func-groups` allows you to control function grouping behavior.
`-no-fnsplit`	`/Qfnsplit-`	Disables function splitting. Function splitting is enabled by the `prof-use` option in phase 3 to improve code locality by splitting routines into different sections: one section to contain the cold or very infrequently executed (cold) code, and one section to contain the rest of the frequently executed (hot)

Linux* and Mac OS* X	Windows*	Effect
		code. You may want to disable function splitting for the following reasons: • Improve debugging capability. In the debug symbol table, it is difficult to represent a split routine, that is, a routine with some of its code in the hot code section and some of its code in the cold code section. • Account for the cases when the profile data does not represent the actual program behavior, that is, when the routine is actually used frequently rather than infrequently. This option is supported on IA-32 architecture for Windows OS and on IA-64 architecture for Windows and Linux OS. It is not supported on other platforms (Intel® 64 architecture, Mac OS X, and Linux on IA-32 architecture). Windows: This option behaves differently on systems based on IA-32 architecture than it does on systems based on IA-64 architecture. IA-32 architecture, Windows OS: • The option completely disables function splitting, placing all the code in one section.

Linux* and Mac OS* X	Windows*	Effect
		IA-64 architecture, Linux and Windows OS: • The option disables the splitting within a routine but enables function grouping, an optimization in which entire routines are placed either in the cold code section or the hot code section. Function grouping does not degrade debugging capability.
-prof-func-groups	/Qprof-func-order	Enables ordering of program routines using profile information when specified with `prof-use` (phase 3). The instrumented program (phase 1) must have been compiled with the `prof-gen` option `srcpos` keyword. Not valid for multi-file compilation with the `ipo` options. Mac OS X: Not supported. IA-64 architecture: Not supported. For more information, see Using Function Ordering, Function Order Lists, Function Grouping, and Data Ordering Optimizations.
-prof-data-order	/Qprof-data-order	Enables ordering of static program data items based on profiling information when specified with `prof-use`. The instrumented program (phase 1) must have been compiled

Linux* and Mac OS* X	Windows*	Effect
		with the `prof-gen` option `srcpos` keyword. Not valid for multi-file compilation with the `ipo` options. Mac OS X: Not supported. For more information, see Using Function Ordering, Function Order Lists, Function Grouping, and Data Ordering Optimizations.
`-prof-src-dir`	`/Qprof-src-dir`	Controls whether full directory information for the source directory path is stored in or read from dynamic-information (dyn) files. When used during phase 1 compilation (`prof-gen`), this determines whether the full path is added into dyn file created during instrumented application execution. When used during `profmerge` or phase 3 compilation (`prof-use`), this determines whether the full path for source file names is used or ignored when reading the dyn or dpi files. Using the default `-prof-src-dir` (Linux and Mac OS X) or `/Qprof-src-dir` (Windows) uses the full directory information and also enables the use of the `prof-src-root` and `prof-src-cwd` options.

Linux* and Mac OS* X	Windows*	Effect
		If you specify `-no-prof-src-dir` (Linux and Mac OS X) or `/Qprof-src-dir` (Windows), only the file name (and not the full path) is stored or used. If you do this, all dyn or dpi files must be in the current directory and the `prof-src-root` and `prof-src-cwd` options are ignored.
`-prof-dir`	`/Qprof-dir`	Specifies the directory in which dynamic information (dyn) files are created in, read from, and stored; otherwise, the dyn files are created in or read from the current directory used during compilation. For example, you can use this option when compiling in phase 1 (`prof-gen` option) to define where dynamic information files will be created when running the instrumented executable in phase 2. You also can use this option when compiling in phase 3 (`prof-use` option) to define where the dynamic information files will be read from and a summary file (dpi) created.
`-prof-src-root` or `-prof-src-cwd`	`/Qprof-src-root` or `/Qprof-src-cwd`	Specifies a directory path prefix for the root directory where the user's application files are stored: • To specify the directory prefix root where source files are stored, specify

Linux* and Mac OS* X	Windows*	Effect
		the `-prof-src-root` (Linux and Mac OS X) or `/Qprof-src-root` (Windows) option. • To use the current working directory, specify the `-prof-src-cwd` (Linux and Mac OS X) or `/Qprof-src-cwd` (Windows) option. This option is ignored if you specify `-no-prof-src-dir` (Linux and Mac OS X) or `/Qprof-src-dir` (Windows).
`-prof-file`	`/Qprof-file`	Specifies file name for profiling summary file. If this option is not specified, the name of the file containing summary information will be pgopti.dpi.
`-prof-gen-sampling`	`/Qprof-gen-sampling`	IA-32 architecture. Prepares application executables for hardware profiling (sampling) and causes the compiler to generate source code mapping information. Mac OS X: This option is not supported.
`-ssp`	`/Qssp`	IA-32 architecture. Enables Software-based Speculative Pre-computation (SSP) optimization. Mac OS X: This option is not supported.

Refer to Quick Reference Lists for a complete listing of the quick reference topics.

Profile an Application

Profiling an application includes the following three phases:

- Instrumentation compilation and linking
- Instrumented execution
- Feedback compilation

This topic provides detailed information on how to profile an application by providing sample commands for each of the three phases (or steps).

1. Instrumentation compilation and linking

   Use `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) to produce an executable with instrumented information included.

Operating System	Commands
Linux and Mac OS X	`ifort -prof-gen -prof-dir/usr/profiled a1.f90 a2.f90 a3.f90` `ifort -oa1 a1.o a2.o a3.o`
Windows	`ifort /Qprof-gen /Qprof-dirc:\profiled a1.f90 a2.f90 a3.f90` `ifort a1.obj a2.obj a3.obj`

   Use the `-prof-dir` (Linux and Mac OS X) or `/Qprof-dir` (Windows) option if the application includes the source files located in multiple directories; using the option insures the profile information is generated in one consistent place. The example commands demonstrate how to combine these options on multiple sources.

   The compiler gathers extra information when you use the `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows) option; however, the extra information is collected to provide support only for specific Intel tools, like the code-coverage tool. If you do not expect to use such tools, do not specify `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows); the extended option does not provide better optimization and could slow parallel compile times.

2. Instrumented execution

Run your instrumented program with a representative set of data to create one or more dynamic information files.

Operating System	Command
Linux and Mac OS X	`./a1.out`
Windows	`a1.exe`

Executing the instrumented applications generates dynamic information file that has a unique name and .dyn suffix. A new dynamic information file is created every time you execute the instrumented program.

You can run the program more than once with different input data.

3. Feedback compilation

Before this step, copy all .dyn and .dpi files into the same directory. Compile and link the source files with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows); the option instructs the compiler to use the generated dynamic information to guide the optimization:

Operating System	Examples
Linux and Mac OS X	`ifort -prof-use -ipo -prof-dir/usr/profiled a1.f90 a2.f90 a3.f90`
Windows	`ifort /Qprof-use /Qipo /Qprof-dirc:\profiled a1.f90 a2.f90 a3.f90`

This final phase compiles and links the sources files using the data from the dynamic information files generated during instrumented execution (phase 2).

In addition to the optimized executable, the compiler produces a pgopti.dpi file.

Most of the time, you should specify the default optimizations,`-O2` (Linux and Mac OS X) or `/O2` (Windows), for phase 1, and specify more advanced optimizations, `-ipo` (Linux) or `/Qipo` (Windows), during the final (phase 3) compilation. For example, the example shown above used`-O2` (Linux and Mac OS X) or `/O2` (Windows) in step 1 and`-ipo` (Linux or Mac OS X) or `/Qipo` (Windows) in step 3.

NOTE. The compiler ignores the `-ipo` or `-ip` (Linux and Mac OS X) or `/Qipo` or `/Qip` (Windows) option during phase 1 with `-prof-gen` (Linux and Mac OS X) or `/Qprof-gen` (Windows).

PGO Tools

PGO Tools Overview

This section describes the tools that take advantage or support the Profile-guided Optimizations (PGO) available in the compiler.

- code coverage tool
- test prioritization tool
- profmerge tool
- proforder tool

In addition to the tools, this section also contains information on using Software-based Speculative Precomputation, which will allow you to optimize applications using profiling- and sampling-feedback methods on IA-32 architectures.

code coverage Tool

The code coverage tool provides software developers with a view of how much application code is exercised when a specific workload is applied to the application. To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations. The major features of the code coverage tool are:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

The information about using the code coverage tool is separated into the following sections:

- code coverage tool Requirements
- Visually Presenting Code Coverage for an Application
- Excluding Code from Coverage Analysis
- Exporting Coverage Data

The tool analyzes static profile information generated by the compiler, as well as dynamic profile information generated by running an instrumented form of the application binaries on the workload. The tool can generate the in HTML-formatted report and export data in both text-, and XML-formatted files. The reports can be further customized to show color-coded, annotated, source-code listings that distinguish between used and unused code.

The code coverage tool is available on IA-32, Intel® 64, and IA-64 architectures on Linux* and Windows*. The tool is available on IA-32 and Intel® 64 architectures on Mac OS* X.

You can use the tool in a number of ways to improve development efficiency, reduce defects, and increase application performance:

- During the project testing phase, the tool can measure the overall quality of testing by showing how much code is actually tested.
- When applied to the profile of a performance workload, the code coverage tool can reveal how well the workload exercises the critical code in an application. High coverage of performance-critical modules is essential to taking full advantage of the Profile-Guided Optimizations that Intel Compilers offer.
- The tool provides an option, useful for both coverage and performance tuning, enabling developers to display the dynamic execution count for each basic block of the application.
- The code coverage tool can compare the profile of two different application runs. This feature can help locate portions of the code in an application that are unrevealed during testing but are exercised when the application is used outside the test space, for example, when used by a customer.

code coverage tool Requirements

To run the code coverage tool on an application, you must have following items:

- The application sources.
- The .spi file generated by the Intel® compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows) options.
- A `pgopti.dpi` file that contains the results of merging the dynamic profile information (.dyn) files, which is most easily generated by the profmerge tool. This file is also generated implicitly by the Intel® compilers when compiling an application with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows) options with available .dyn and .dpi files.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

Using the Tool

The tool uses the following syntax:

Tool Syntax
`codecov [-codecov_option]`

where `-codecov_option` is one or more optional parameters specifying the tool option passed to the tool. The available tool options are listed in code coverage tools Options (below). If you do not use any additional tool options, the tool will provide the top-level code coverage for the entire application.

In general, you must perform the following steps to use the code coverage tool:

1. Compile the application using `-prof-gen=srcpos` (Linux and Mac OS X) or `/Qprof-gen:srcpos` (Windows).

   This step generates an instrumented executable and a corresponding static profile information (pgopti.spi) file.

2. Run the instrumented application.

   This step creates the dynamic profile information (.dyn) file Each time you run the instrumented application, the compiler generates a unique .dyn file either in the current directory or the directory specified in by prof_dir.

3. Use the profmerge tool to merge all the .dyn files into one .dpi (`pgopti.dpi`) file.

   This step consolidates results from all runs and represents the total profile information for the application, generates an optimized binary, and creates the dpi file needed by the code coverage tool.

   You can use the profmerge tool to merge the .dyn files into a .dpi file without recompiling the application. The profmerge tool can also merge multiple .dpi files into one .dpi file using the `profmerge -a` option. Select the name of the output .dpi file using the `profmerge -prof_dpi` option.

CAUTION. The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code.

4. Run the code coverage tool. (The valid syntax and tool options are shown below.)

   This step creates a report or exported data as specified. If no other options are specified, the code coverage tool creates a single HTML file (CODE_COVERAGE.HTML) and a sub-directory (CodeCoverage) in the current directory. Open the file in a web browser to view the reports.

NOTE. Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

The code coverage tool allows you to name the project and specify paths to specific, necessary files. The following example demonstrates how to name a project and specify .dpi and .spi files to use:

Example: specify .dpi and .spi files

```
codecov -prj myProject -spi pgopti.spi -dpi pgopti.dpi
```

The tool can add a contact name and generate an email link for that contact at the bottom of each HTML page. This provides a way to send an electronic message to the named contact. The following example demonstrates how to add specify a contact and the email links:

Example: add contact information

```
codecov -prj myProject -mname JoeSmith -maddr js@company.com
```

This following example demonstrates how to use the tool to specify the project name, specify the dynamic profile information file, and specify the output format and file name.

Example: export data to text

```
codecov -prj test1 -dpi test1.dpi -txtbcvrg test1_bcvrg.txt
```

code coverage tool Options

The tool uses the options listed in the table:

Option	Default	Description
`-bcolor` *`color`*	#FFFF99	Specifies the HTML color name for code in the uncovered blocks.
`-beginblkdsbl` *`string`*		Specifies the comment that marks the beginning of the code fragment to be ignored by the coverage tool.

Option	Default	Description
-ccolor *color*	#FFFFFF	Specifies the HTML color name or code of the covered code.
-comp *file*		Specifies the file name that contains the list of files being (or not) displayed.
-counts		Generates dynamic execution counts.
-demang		Demangles both function names and their arguments.
-dpi *file*	pgopti.dpi	Specifies the file name of the dynamic profile information file (.dpi).
-endblkdsbl *string*		Specifies the comment that marks the end of the code fragment to be ignored by the coverage tool.
-fcolor *color*	#FFCCCC	Specifies the HTML color name for code of the uncovered functions.
-help, -h		Prints tool option descriptions.
-icolor *color*	#FFFFFF	Specifies the HTML color name or code of the information lines, such as basic-block markers and dynamic counts.
-maddr *string*	Nobody	Sets the email address of the web-page owner

Option	Default	Description
-mname *string*	Nobody	Sets the name of the web-page owner.
-nopartial		Treats partially covered code as fully covered code.
-nopmeter		Turns off the progress meter. The meter is enabled by default.
-onelinedsbl string		Specifies the comment that marks individual lines of code or the whole functions to be ignored by the coverage tool.
-pcolor *color*	#FAFAD2	Specifies the HTML color name or code of the partially covered code.
-prj *string*		Sets the project name.
-ref		Finds the differential coverage with respect to ref_dpi_file.
-spi *file*	pgopti.spi	Specifies the file name of the static profile information file (.spi).
-srcroot *dir*		Specifies a different top level project directory than was used during compiler instrumentation run to use for relative paths to source files in place of absolute paths.

Option	Default	Description
-txtbcvrg *file*		Export block-coverage for covered functions as text format. The file parameter must be in the form of a valid file name.
-txtbcvrgfull *file*		Export block-coverage for entire application in text and HTML formats. The file parameter must be in the form of a valid file name.
-txtdcg *file*		Generates the dynamic call-graph information in text format. The file parameter must be in the form of a valid file name.
-txtfcvrg *file*		Export function coverage for covered function in text format. The file parameter must by in the form of a valid file name.
-ucolor *color*	#FFFFFF	Specifies the HTML color name or code of the unknown code.
-xcolor *color*	#90EE90	Specifies the HTML color of the code ignored.
-xmlbcvrg *file*		Export the block-coverage for the covered function in XML format. The file parameter must by in the form of a valid file name.

Option	Default	Description
`-xmlbcvrgfull` *file*		Export function coverage for entire application in XML format in addition to HTML output. The file parameter must be in the form of a valid file name.
`-xmlfcvrg` *file*		Export function coverage for covered function in XML format. The file parameter must be in the form of a valid file name.

Visually Presenting Code Coverage for an Application

Based on the profile information collected from running the instrumented binaries when testing an application, the Intel® compiler will create HTML-formatted reports using the code coverage tool. These reports indicate portions of the source code that were or were not exercised by the tests. When applied to the profile of the performance workloads, the code coverage information shows how well the training workload covers the application's critical code. High coverage of performance-critical modules is essential to taking full advantage of the profile-guided optimizations.

The code coverage tool can create two levels of coverage:

- Top level (for a group of selected modules)
- Individual module source views

Top Level Coverage

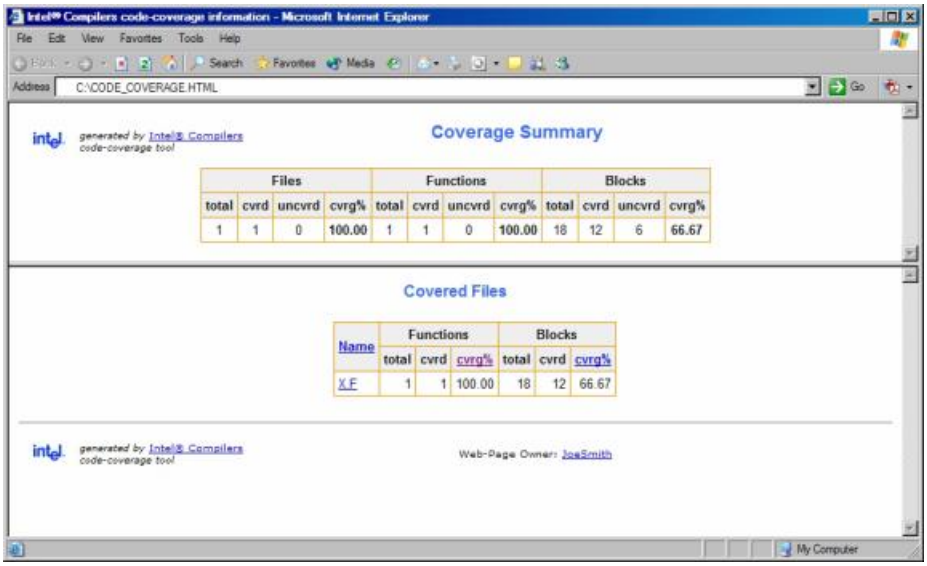
The top-level coverage reports the overall code coverage of the modules that were selected. The following options are provided:

- Select the modules of interest
- For the selected modules, the tool generates a list with their coverage information. The information includes the total number of functions and blocks in a module and the portions that were covered.

- By clicking on the title of columns in the reported tables, the lists may be sorted in ascending or descending order based on:
  - basic block coverage
  - function coverage
  - function name

By default, the code coverage tool generates a single HTML file (CODE_COVERAGE.HTML) and a subdirectory (CodeCoverage) in the current directory. The HTML file defines a frameset to display all of the other generated reports. Open the HTML file in a web-browser. The tool places all other generated report files in a CodeCoverage subdirectory.

If you choose to generate the html-formatted version of the report, you can view coverage source of that particular module directly from a browser. The following figure shows the top-level coverage report.



The coverage tool creates a frame set that allows quick browsing through the code to identify uncovered code. The top frame displays the list of uncovered functions while the bottom frame displays the list of covered functions. For uncovered functions, the total number of basic blocks of each function is also displayed. For covered functions, both the total number of blocks and the number of covered blocks as well as their ratio (that is, the coverage rate) are displayed.

For example, 66.67(4/6) indicates that four out of the six blocks of the corresponding function were covered. The block coverage rate of that function is thus 66.67%. These lists can be sorted based on the coverage rate, number of blocks, or function names. Function names are linked to the position in source view where the function body starts. So, just by one click, you can see the least-covered function in the list and by another click the browser displays the body of the function. You can scroll down in the source view and browse through the function body.

Individual Module Source View

Within the individual module source views, the tool provides the list of uncovered functions as well as the list of covered functions. The lists are reported in two distinct frames that provide easy navigation of the source code. The lists can be sorted based on:

- Number of blocks within uncovered functions
- Block coverage in the case of covered functions
- Function names

Setting the Coloring Scheme for the Code Coverage

The tool provides a visible coloring distinction of the following coverage categories: covered code, uncovered basic blocks, uncovered functions, partially covered code, and unknown code. The default colors that the tool uses for presenting the coverage information are shown in the tables that follows:

Category	Default	Description
Covered code	#FFFFFF	Indicates code was exercised by the tests. You can override the default color with the `-ccolor` tool option.
Uncovered basic block	#FFFF99	Indicates the basic blocks that were not exercised by any of the tests. However, these blocks were within functions that were executed

Category	Default	Description
		during the tests. You can override the default color with the `-bcolor` tool option.
Uncovered function	#FFCCCC	Indicates functions that were never called during the tests. You can override the default color with the `-fcolor` tool option.
Partially covered code	#FAFAD2	Indicates that more than one basic block was generated for the code at this position. Some of the blocks were covered and some were not. You can override the default color with the `-pcolor` tool option.
Ignored code	#90EE90	Indicates code that was specifically marked to be ignored. You can override this default color using the `-xcolor` tool option.
Information lines	#FFFFFF	Indicates basic-block markers and dynamic counts. You can override the default color with the `-icolor` tool option.
Unknown	#FFFFFF	Indicates that no code was generated for this source line. Most probably, the source at this position is a comment, a header-file inclusion, or a variable declaration. You can override the default color with the `-ucolor` tool option.

The default colors can be customized to be any valid HTML color name or hexadecimal value using the options mentioned for each coverage category in the table above.

For code coverage colored presentation, the coverage tool uses the following heuristic: source characters are scanned until reaching a position in the source that is indicated by the profile information as the beginning of a basic block. If the profile information for that basic block indicates that a coverage category changes, then the tool changes the color corresponding to the coverage condition of that portion of the code, and the coverage tool inserts the appropriate color change in the HTML-formatted report files.

NOTE. You need to interpret the colors in the context of the code. For instance, comment lines that follow a basic block that was never executed would be colored in the same color as the uncovered blocks.

Dynamic Counters

The coverage tool can be configured to generate the information about the dynamic execution counts. This ability can display the dynamic execution count of each basic block of the application and is useful for both coverage and performance tuning.

The custom configuration requires using the `-counts` option. The counts information is displayed under the code after a "^" sign precisely under the source position where the corresponding basic block begins.

If more than one basic block is generated for the code at a source position (for example, for macros), then the total number of such blocks and the number of the blocks that were executed are also displayed in front of the execution count. For example, line 11 in the code is an `IF` statement:

Example

```
11   IF ((N .EQ. 1).OR. (N .EQ. 0))
     ^ 10 (1/2)
12      PRINT N
        ^ 7
```

The coverage lines under code lines 11 and 12 contain the following information:

- The `IF` statement in line 11 was executed 10 times.
- Two basic blocks were generated for the `IF` statement in line 11.

- Only one of the two blocks was executed, hence the partial coverage color.
- Only seven out of the ten times variable n had a value of 0 or 1.

In certain situations, it may be desirable to consider all the blocks generated for a single source position as one entity. In such cases, it is necessary to assume that all blocks generated for one source position are covered when at least one of the blocks is covered. This assumption can be configured with the `-nopartial` option. When this option is specified, decision coverage is disabled, and the related statistics are adjusted accordingly. The code lines 11 and 12 indicate that the `print` statement in line 12 was covered. However, only one of the conditions in line 11 was ever true. With the `-nopartial` option, the tool treats the partially covered code (like the code on line 11) as covered.

Differential Coverage

Using the code coverage tool, you can compare the profiles from two runs of an application: a reference run and a new run identifying the code that is covered by the new run but not covered by the reference run. Use this feature to find the portion of the applications code that is not covered by the applications tests but is executed when the application is run by a customer. It can also be used to find the incremental coverage impact of newly added tests to an applications test space.

Generating Reference Data

Create the dynamic profile information for the reference data, which can be used in differential coverage reporting later, by using the `-ref` option. The following command demonstrate a typical command for generating the reference data:

Example: generating reference data
`codecov -prj Project_Name -dpi customer.dpi -ref appTests.dpi`

The coverage statistics of a differential-coverage run shows the percentage of the code exercised on a new run but missed in the reference run. In such cases, the tool shows only the modules that included the code that was not covered. Keep this in mind when viewing the coloring scheme in the source views.

The code that has the same coverage property (covered or not covered) on both runs is considered as covered code. Otherwise, if the new run indicates that the code was executed while in the reference run the code was not executed, then the code is treated as uncovered. On the other hand, if the code is covered in the reference run but not covered in the new run, the differential-coverage source view shows the code as covered.

Running Differential Coverage

To run the code coverage tool for differential coverage, you must have the application sources, the .spi file, and the .dpi file, as described in the code coverage tool Requirements section (above).

Once the required files are available, enter a command similar to the following begin the process of differential coverage analysis:

Example
`codecov -prj Project_Name -spi pgopti.spi -dpi pgopti.dpi`

Specify the .dpi and .spi files using the `-spi` and `-dpi` options.

Excluding Code from Coverage Analysis

The code coverage tool allows you to exclude portions of your code from coverage analysis. This ability can be useful during development; for example, certain portions of code might include functions used for debugging only. The test case should not include tests for functionality that will unavailable in the final application.

Another example of code that can be excluded is code that might be designed to deal with internal errors unlikely to occur in the application. In such cases, not having a test case lack of a test case is preferred. You might want to ignore infeasible (dead) code in the coverage analysis. The code coverage tool provides several options for marking portions of the code infeasible (dead) and ignoring the code at the file level, function level, line level, and arbitrary code boundaries indicated by user-specific comments. The following sections explain how to exclude code at different levels.

Including and Excluding Coverage at the File Level

The code coverage tool provides the ability to selectively include or exclude files for analysis. Create a component file and add the appropriate string values that indicate the file and directory name for code you want included or excluded. Pass the file name as a parameter of the `-comp` option. The following example shows the general command:

Example: specifying a component file
`codecov -comp file`

where *`file`* is the name of a text file containing strings that ask as file and directory name masks for including and excluding file-level analysis. For example, assume that the following:

- You want to include all files with the string "source" in the file name or directory name.

- You create a component text file named `myComp.txt` with the selective inclusion string.

Once you have a component file, enter a command similar to the following:

Example

```
codecov -comp myComp.txt
```

In this example, individual files name including the string "source" (like source1.f and source2.f) and files in directories where the name contains the string "source" (like source/file1.f and source2\file2.f) are include in the analysis.

Excluding files is done in the same way; however, the string must have a tilde (~) prefix. The inclusion and exclusion can be specified in the same component file.

For example, assume you want to analyze all individual files or files contained in a directory where the name included the string "source", and you wanted to exclude all individual file and files contained in directories where the name included the string "skip". You would add content similar to the following to the component file (`myComp.txt`) and pass it to the `-comp` option:

Example: inclusion and exclusion strings

```
source
~skip
```

Entering the `codecov -comp myComp.txt` command with both instructions in the component file will instruct the tool to include individual files where the name contains "source" (like source1.f and source2.f) and directories where the name contains "source" (like source/file1.f and source2\file2.f), and exclude any individual files where the name contains "skip" (like skipthis1.f and skipthis2.f) or directories where the name contains "skip" (like skipthese1\debug1.f and skipthese2\debug2.f).

Excluding Coverage at the Line and Function Level

You can mark individual lines for exclusion my passing string values to the `-onelinedsbl` option. For example, assume that you have some code similar to the following:

Sample code

```
print*, "ERROR: n  = ", n  ! NO_COVER
print*, "       n2 = ", n2  ! INF IA-32 architecture
```

If you wanted to exclude all functions marked with the comments NO_COVER or INF IA-32 architecture, you would enter a command similar to the following.

Example

```
codecov -onelinedsbl NO_COVER -onelinedsbl "INF IA-32 architecture"
```

You can specify multiple exclusion strings simultaneously, and you can specify any string values for the markers; however, you must remember the following guidelines when using this option:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

An entire function can be excluded from coverage analysis using the same methods. For example, the following function will be ignored from the coverage analysis when you issue example command shown above.

Sample code

```
subroutine dumpInfo (n)
integer n ! NO_COVER
...
end subroutine
```

Additionally, you can use the code coverage tool to color the infeasible code with any valid HTML color code by combining the `-onelinedsbl` and `-xcolor` options. The following example commands demonstrate the combination:

Example: combining tool options

```
codecov -onelinedsbl INF -xcolor lightgreen
codecov -onelinedsbl INF -xcolor #CCFFCC
```

Excluding Code by Defining Arbitrary Boundaries

The code coverage tool provides the ability to arbitrarily exclude code from coverage analysis. This feature is most useful where the excluded code either occur inside of a function or spans several functions.

Use the -beginblkdsbl and -endblkdsbl options to mark the beginning and end, respectively, of any arbitrarily defined boundary to exclude code from analysis. Remember the following guidelines when using these options:

- Inline comments must occur at the end of the statement.
- The string must be a part of an inline comment.

For example assume that you have the following code:

Sample code

```
integer n, n2
n = 123
n2 = n*n
if (n2 .lt. 0) then
! /* BINF */
  print*, "ERROR: n  = ", n
  print*, "       n2 = ", n2
! // EINF
else if (n2 .eq. 0) then
  print*, "zero: n = ", n, " n2 = ", n2
else
  print*, "positive: n = ", n, " n2 = ", n2
endif
end
```

The following example commands demonstrate how to use the -beginblkdsbl option to mark the beginning and the -endblkdsbl option to mark the end of code to exclude from the sample shown above.

Example: arbitrary code marker commands

```
codecov -xcolor #ccFFCC -beginblkdsbl BINF -endblkdsbl EINF
codecov -xcolor #ccFFCC -beginblkdsbl "BEGIN_INF" -endblkdsbl "END_INF"
```

Notice that you can combine these options in combination with the `-xcolor` option.

Exporting Coverage Data

The code coverage tool provides specific options to extract coverage data from the dynamic profile information (.dpi files) that result from running instrumented application binaries under various workloads. The tool can export the coverage data in various formats for post-processing and direct loading into databases: the default HTML, text, and XML. You can choose to export data at the function and basic block levels.

There are two basic methods for exporting the data: quick export and combined export. Each method has associated options supported by the tool

- **Quick export:** The first method is to export the data coverage to text- or XML-formatted files without generating the default HTML report. The application sources need not be present for this method. The code coverage tool creates reports and provides statistics only about the portions of the application executed. The resulting analysis and reporting occurs quickly, which makes it practical to apply the coverage tool to the dynamic profile information (the .dpi file) for every test case in a given test space instead of applying the tool to the profile of individual test suites or the merge of all test suites. The `-xmlfcvrg`, `-txtfcvrg`, `-xml-bcvrg` and `-txtbcvrg` options support the first method.
- **Combined export:** The second method is to generate the default HTML and simultaneously export the data to text- and XML-formatted files. This process is slower than first method since the application sources are parsed and reports generated. The `-xmlbcvrgfull` and `-txtbcvrgfull` options support the second method.

These export methods provide the means to quickly extend the code coverage reporting capabilities by supplying consistently formatted output from the code coverage tool. You can extend these by creating additional reporting tools on top of these report files.

Quick Export

The profile of covered functions of an application can be exported quickly using the `-xmlfcvrg`, `-txtfcvrg`, `-xmlbcvrg`, and `-txtbcvrg` options. When using any of these options, specify the output file that will contain the coverage report. For example, enter a command similar to the following to generate a report of covered functions in XML formatted output:

Example: quick export of function data

```
codecov -prj test1 -dpi test1.dpi -xmlfcvrg test1_fcvrg.xml
```

The resulting report will show how many times each function was executed and the total number of blocks of each function together with the number of covered blocks and the block coverage of each function. The following example shows some of the content of a typical XML report.

XML-formatted report example

```
<PROJECT name = "test1">
   <MODULE name = "D:\SAMPLE.F">
      <FUNCTION name="f0" freq="2">
         <BLOCKS total="6" covered="5" coverage="83.33%"></BLOCKS>
      </FUNCTION>
      ...
   </MODULE>
   <MODULE name = "D:\SAMPLE2.F">
      ...
   </MODULE>
</PROJECT>
```

In the above example, we note that function f0, which is defined in file sample.f, has been executed twice. It has a total number of six basic blocks, five of which are executed, resulting in an 83.33% basic block coverage.

You can also export the data in text format using the `-txtfcvrg` option. The generated text report, using this option, for the above example would be similar to the following example:

Text-formatted report example

```
Covered Functions in File: "D:\SAMPLE.F"
"f0"    2       6       5        83.33
"f1"    1       6       4        66.67
"f2"    1       6       3        50.00
...
```

In the text formatted version of the report, the each line of the report should be read in the following manner:

Column 1	Column 2	Column 3	Column 4	Column 5
function name	execution frequency	line number of the start of the function definition	column number of the start of the function definition	percentage of basic-block coverage of the function

Additionally, the tool supports exporting the block level coverage data using the `-xmlbcvrg` option. For example, enter a command similar to the following to generate a report of covered blocks in XML formatted output:

Example: quick export of block data to XML

```
codecov -prj test1 -dpi test1.dpi -xmlbcvrg test1_bcvrg.xml
```

The example command shown above would generate XML-formatted results similar to the following:

XML-formatted report example

```
<PROJECT name = "test1">
  <MODULE name = "D:\SAMPLE.cF90">
    <FUNCTION name="f0" freq="2">
    ...
      <BLOCK line="11" col="2">
        <INSTANCE id="1" freq="1"> </INSTANCE>
      </BLOCK>
      <BLOCK line="12" col="3">
        <INSTANCE id="1" freq="2"> </INSTANCE>
        <INSTANCE id="2" freq="1"> </INSTANCE>
      </BLOCK>
```

In the sample report, notice that one basic block is generated for the code in function f0 at the line 11, column 2 of the file sample.f90. This particular block has been executed only once. Also notice that there are two basic blocks generated for the code that starts at line 12, column

3 of file. One of these blocks, which has id = 1, has been executed two times, while the other block has been executed only once. A similar report in text format can be generated through the `-txtbcvrg` option.

Combined Exports

The code coverage tool has also the capability of exporting coverage data in the default HTML format while simultaneously generating the text- and XML-formatted reports.

Use the `-xmlbcvrgfull` and `-txtbcvrgfull` options to generate reports in all supported formatted in a single run. These options export the basic-block level coverage data while simultaneously generating the HTML reports. These options generate more complete reports since they include analysis on functions that were not executed at all. However, exporting the coverage data using these options requires access to application source files and take much longer to run.

Dynamic Call Graphs

Using the `-txtdcg` option the tool can provide detailed information about the dynamic call graphs in an application. Specify an output file for the dynamic call-graph report. The resulting call graph report contains information about the percentage of static and dynamic calls (direct, indirect, and virtual) at the application, module, and function levels.

test prioritization Tool

The test prioritization tool enables the profile-guided optimizations on IA-32, Intel® 64, and IA-64 architectures, on Linux* and Windows*, to select and prioritize test for an application based on prior execution profiles. The tool is available on IA-32 and Intel® 64 architectures on Mac OS* X.

The tool offers a potential of significant time saving in testing and developing large-scale applications where testing is the major bottleneck.

Development often requires changing applications modules. As applications change, developers can have a difficult time retaining the quality of their functional and performance tests so they are current and on-target. The test prioritization tool lets software developers select and prioritize application tests as application profiles change.

The information about the tool is separated into the following sections:

- Features and benefits
- Requirements and syntax
- Usage model
- Tool options
- Running the tool

Features and Benefits

The test prioritization tool provides an effective testing hierarchy based on the code coverage for an application. The following list summarizes the advantages of using the tool:

- Minimizing the number of tests that are required to achieve a given overall coverage for any subset of the application: the tool defines the smallest subset of the application tests that achieve exactly the same code coverage as the entire set of tests.
- Reducing the turn-around time of testing: instead of spending a long time on finding a possibly large number of failures, the tool enables the users to quickly find a small number of tests that expose the defects associated with the regressions caused by a change set.
- Selecting and prioritizing the tests to achieve certain level of code coverage in a minimal time based on the data of the tests' execution time.

See Understanding Profile-guided Optimization and Example of Profile-guided Optimization for general information on creating the files needed to run this tool.

test prioritization Tool Requirements

The test prioritization tool needs the following items to work:

- The .spi file generated by Intel® compilers when compiling the application for the instrumented binaries with the `-prof-gen=srcpos` (Linux* and Mac OS* X) or `/Qprof-gen:srcpos` (Windows*) option.
- The .dpi files generated by the profmerge tool as a result of merging the dynamic profile information .dyn files of each of the application tests. (Run the profmerge tool on all .dyn files that are generated for each individual test and name the resulting .dpi in a fashion that uniquely identifies the test.)

CAUTION. The profmerge tool merges all .dyn files that exist in the given directory. Make sure unrelated .dyn files, which may remain from unrelated runs, are not present. Otherwise, the profile information will be skewed with invalid profile data, which can result in misleading coverage information and adverse performance of the optimized code;

- User-generated file containing the list of tests to be prioritized. For successful instrumented code run, you should:
- Name each test .dpi file so the file names uniquely identify each test.

- Create a .dpi list file, which is a text file that contains the names of all .dpi test files.

  Each line of the .dpi list file should include one, and only one .dpi file name. The name can optionally be followed by the duration of the execution time for a corresponding test in the dd:hh:mm:ss format.

  For example: `Test1.dpi 00:00:60:35` states that Test1 lasted 0 days, 0 hours, 60 minutes and 35 seconds.

  The execution time is optional. However, if it is not provided, then the tool will not prioritize the test for minimizing execution time. It will prioritize to minimize the number of tests only.

The tool uses the following general syntax:

Tool Syntax
`tselect -dpi_list` *file*

`-dpi_list` is a required tool option that sets the path to the list file containing the list of the all .dpi files. All other tool commands are optional.

NOTE. Windows* only: Unlike the compiler options, which are preceded by forward slash ("/"), the tool options are preceded by a hyphen ("-").

Usage Model

The following figure illustrates a typical test prioritization tool usage model.

test prioritization tool Options

The tool uses the options that are listed in the following table:

Option	Default	Description
`-help`		Prints tool option descriptions.
`-dpi_list` *file*		Required. Specifies the file name of the file that contains the names of the dynamic

Option	Default	Description
		profile information (`.dpi`) files. Each line of the file must contain only one `.dpi` file name, which can be optionally followed by its execution time. The name must uniquely identify the test.
`-spi` *file*	`pgopti.spi`	Specifies the file name of the static profile information file (.SPI).
`-o` *file*		Specifies the file name of the output report file.
`-comp` *file*		Specifies the file name that contains the list of files of interest.
`-cutoff` *file*		Instructs the tool to terminate when the cumulative block coverage reaches a preset percentage, as specified by `value`, of pre-computed total coverage. `value` must be greater than 0.0 (for example, 99.00) but not greater than 100. `value` can be set to 100.
`-nototal`		Instructs the tool to ignore the pre-compute total coverage process.
`-mintime`		Instructs the tool to minimize testing execution time. The execution time of each test must be provided on the

Option	Default	Description
		same line of `dpi_list` file, after the test name in dd:hh:mm:ss format.
`-srcbasedir` *dir*		Specifies a different top level project directory than was used during compiler instrumentation run with the `prof-src-root` compiler option to support relative paths to source files in place of absolute paths.
`-verbose`		Instructs the tool to generate more logging information about program progress.

Running the tool

The following steps demonstrate one simple example for running the tool on IA-32 architectures.

1. Specify the directory by entering a command similar to the following:

Example
`set prof-dir=c:\myApp\prof-dir`

2. Compile the program and generate instrumented binary by issuing commands similar to the following:

Operating System	Command
Linux and Mac OS X	`ifort -prof-gen=srcpos myApp.f90`
Windows	`ifort /Qprof-gen:srcpos myApp.f90`

This commands shown above compiles the program and generates instrumented binary `myApp` as well as the corresponding static profile information `pgopti.spi`.

3. Make sure that unrelated `.dyn` files are not present by issuing a command similar to the following:

Example
`rm prof-dir \*.dyn`

4. Run the instrumented files by issuing a command similar to the following:

Example
`myApp < data1`

   The command runs the instrumented application and generates one or more new dynamic profile information files that have an extension `.dyn` in the directory specified by the `-prof-dir` step above.

5. Merge all `.dyn` file into a single file by issuing a command similar to the following:

Example
`profmerge -prof_dpi Test1.dpi`

   The profmerge tool merges all the .dyn files into one file (`Test1.dpi`) that represents the total profile information of the application on Test1.

6. Again make sure there are no unrelated .dyn files present a second time by issuing a command similar to the following:

Example
`rm prof-dir \*.dyn`

7. Run the instrumented application and generate one or more new dynamic profile information files that have an extension .dyn in the directory specified the `prof-dir` step above by issuing a command similar to the following:

Example
`myApp < data2`

8. Merge all .dyn files into a single file, by issuing a command similar to the following

Example
`profmerge -prof_dpi Test2.dpi`

At this step, the profmerge tool merges all the .dyn files into one file (`Test2.dpi`) that represents the total profile information of the application on Test2.

9. Make sure that there are no unrelated .dyn files present for the final time, by issuing a command similar to the following:

Example
`rm prof-dir \*.dyn`

10. Run the instrumented application and generates one or more new dynamic profile information files that have an extension .dyn in the directory specified by `-prof-dir` by issuing a command similar to the following:

Example
`myApp < data3`

11. Merge all .dyn file into a single file, by issuing a command similar to the following:

Example
`profmerge -prof_dpi Test3.dpi`

At this step, the profmerge tool merges all the .dyn files into one file (`Test3.dpi`) that represents the total profile information of the application on Test3.

12. Create a file named `tests_list` with three lines. The first line contains `Test1.dpi`, the second line contains `Test2.dpi`, and the third line contains `Test3.dpi`.

Tool Usage Examples

When these items are available, the test prioritization tool may be launched from the command line in `prof-dir` directory as described in the following examples.

Example 1: Minimizing the Number of Tests

The following example describes how minimize the number of test runs.

Example Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi
```

where the `-spi` option specifies the path to the .spi file.

The following sample output shows typical results.

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
 num  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  --------  --------  --------  -------------------
   1   87.50     45.65     37.50    Test3.dpi
   2   100.50    52.17     50.00    Test2.dpi
```

In this example, the results provide the following information:

- By running all three tests, we achieve 52.17% block coverage and 50.00% function coverage.
- Test3 by itself covers 45.65% of the basic blocks of the application, which is 87.50% of the total block coverage that can be achieved from all three tests.
- By adding Test2, we achieve a cumulative block coverage of 52.17% or 100% of the total block coverage of Test1, Test2, and Test3.
- Elimination of Test1 has no negative impact on the total block coverage.

Example 2: Minimizing Execution Time

Suppose we have the following execution time of each test in the `tests_list` file:

Sample Output

```
Test1.dpi 00:00:60:35
Test2.dpi 00:00:10:15
Test3.dpi 00:00:30:45
```

The following command minimizes the execution time by passing the `-mintime` option:

Sample Syntax

```
tselect -dpi_list tests_list -spi pgopti.spi -mintime
```

The following sample output shows possible results:

Sample Output

```
Total number of tests   =  3
Total block coverage    ~  52.17
Total function coverage ~  50.00
Total execution time    =  1:41:35
 num  elaspedTime  %RatCvrg  %BlkCvrg  %FncCvrg  Test Name @ Options
 ---  -----------  --------  --------  --------  ------------------
   1   10:15        75.00     39.13     25.00    Test2.dpi
   2   41:00        100.00    52.17     50.00    Test3.dpi
```

In this case, the results indicate that the running all tests sequentially would require one hour, 45 minutes, and 35 seconds, while the selected tests would achieve the same total block coverage in only 41 minutes.

The order of tests is based on minimizing time (first Test2, then Test3) could be different than when prioritization is done based on minimizing the number of tests. See Example 1 shown above: first Test3, then Test2. In Example 2, Test2 is the test that gives the highest coverage per execution time, so Test2 is picked as the first test to run.

Using Other Options

The `-cutoff` option enables the tool to exit when it reaches a given level of basic block coverage. The following example demonstrates how to the option:

Example

```
tselect -dpi_list tests_list -spi pgopti.spi -cutoff 85.00
```

If the tool is run with the cutoff value of 85.00, as in the above example, only Test3 will be selected, as it achieves 45.65% block coverage, which corresponds to 87.50% of the total block coverage that is reached from all three tests.

The tool does an initial merging of all the profile information to figure out the total coverage that is obtained by running all the tests. The `-nototal` option enables you to skip this step. In such a case, only the absolute coverage information will be reported, as the overall coverage remains unknown.

profmerge and proforder Tools

profmerge Tool

Use the profmerge tool to merge dynamic profile information (.dyn) files and any specified summary files (.dpi). The compiler executes profmerge automatically during the feedback compilation phase when you specify `-prof-use` (Linux* and Mac OS* X) or `/Qprof-use` (Windows*).

The command-line usage for profmerge is as follows:

Syntax
`profmerge [-prof_dir dir_name]`

The tool merges all .dyn files in the current directory, or the directory specified by `-prof_dir`, and produces a summary file: `pgopti.dpi`.

NOTE. The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options ompiler options (for example `-prof-dir` or `/Qprof-dir`) to join words. Also, on Windows* OS systems, the tool options are preceded by a hyphen ("-") unlike Windows compiler options, which are preceded by a forward slash ("/").

You can use profmerge tool to merge .dyn files into a .dpi file without recompiling the application. Thus, you can run the instrumented executable file on multiple systems to generate dyn files, and optionally use profmerge with the `-prof_dpi` option to name each summary dpi file created from the multiple dyn files.

Since the profmerge tool merges all the .dyn files that exist in the given directory, make sure unrelated .dyn files are not present; otherwise, profile information will be based on invalid profile data, which can negatively impact the performance of optimized code.

profmerge Options

The profmerge tool supports the following options:

Tool Option	Description
`-help`	Lists supported options.
`-nologo`	Disables version information. This option is supported on Windows* only.
`-exclude_funcs` *functions*	Excludes functions from the profile. The list items must be separated by a comma (","); you can use a period (".") as a wild card character in function names.
`-prof_dir` *dir*	Specifies the directory from which to read .dyn and .dpi files , and write the .dpi file. Alternatively, you can set the environment variable PROF_DIR.
`-prof_dpi` *file*	Specifies the name of the .dpi file being generated.
`-prof_file` *file*	Merges information from file matching: dpi_file_and_dyn_tag.
`-dump`	Displays profile information.
`-src_old` *dir* `-src_new` *dir*	Changes the directory path stored within the .dpi file.
`-src_no_dir`	Uses only the file name and not the directory name when reading dyn/dpi records. If you specify `-src_no_dir`, the directory name of the source file will be ignored when deciding

Tool Option	Description
	which profile data records correspond to a specific application routine, and the `-src-root` option is ignored.
`-src-root` *dir*	Specifies a directory path prefix for the root directory where the user's application files are stored. This option is ignored if you specify `-src_no_dir`.
`-a` *file1.dpi ... fileN.dpi*	Specifies and merges available .dpi files.
`-verbose`	Instructs the tool to display full information during merge.
`-weighted`	Instructs the tool to apply an equal weighting (regardless of execution times) to the dyn file values to normalize the data counts. This keyword is useful when the execution runs have different time durations and you want them to be treated equally.

Relocating source files using profmerge

The Intel® compiler uses the full path to the source file for each routine to look up the profile summary information associated with that routine. By default, this prevents you from:

- Using the profile summary file (.dpi) if you move your application sources.
- Sharing the profile summary file with another user who is building identical application sources that are located in a different directory.

You can disable the use of directory names when reading dyn/dpi file records by specifying the `profmerge` option `-src_no_dir`. This `profmerge` option is the same as the compiler option `-no-prof-src-dir` (Linux and Mac OS X) and `/Qprof-src-dir-` (Windows).

To enable the movement of application sources, as well as the sharing of profile summary files, you can use the `profmerge` option `-src-root` to specify a directory path prefix for the root directory where the application files are stored. Alternatively, you can specify the option pair `-src_old -src_new` to modify the data in an existing summary dpi file. For example:

Example: relocation command syntax
`profmerge -prof_dir <dir1> -src_old <dir2> -src_new <dir3>`

where *<dir1>* is the full path to dynamic information file (.dpi), *<dir2>* is the old full path to source files, and *<dir3>* is the new full path to source files. The example command (above) reads the `pgopti.dpi` file, in the location specified in *<dir1>*. For each routine represented in the `pgopti.dpi` file, whose source path begins with the *<dir2>* prefix, profmerge replaces that prefix with *<dir3>*. The `pgopti.dpi` file is updated with the new source path information.

You can run profmerge more than once on a given `pgopti.dpi` file. For example, you may need to do this if the source files are located in multiple directories:

Operating System	Command Examples
Linux and Mac OS X	`profmerge -prof_dir -src_old /src/prog_1 -src_new /src/prog_2` `profmerge -prof_dir -src_old /proj_1 -src_new /proj_2`
Windows	`profmerge -src_old "c:/program files" -src_new "e:/program files"` `profmerge -src_old c:/proj/application -src_new d:/app`

In the values specified for `-src_old` and `-src_new`, uppercase and lowercase characters are treated as identical. Likewise, forward slash (`/`) and backward slash (`\`) characters are treated as identical.

NOTE. Because the source relocation feature of profmerge modifies the `pgopti.dpi` file, consider making a backup copy of the file before performing the source relocation.

proforder Tool

The proforder tool is used as part of the feedback compilation phase, to improve program performance. Use proforder to generate a function order list for use with the `/ORDER` linker option. The tool uses the following syntax:

Syntax
`proforder [-prof_dir dir] [-o file]`

where *dir* is the directory containing the profile files (`.dpi` and `.spi`), and *file* is the optional name of the function order list file. The default name is `proford.txt` .

NOTE. The spelling of tools options may differ slightly from compiler options. Tools options use an underscore (for example -prof_dir) instead of the hyphen used by compiler options (for example `-prof-dir` or `/Qprof-dir`) to join words. Also, on Windows* OS systems, the tool options are preceded by a hyphen ("-") unlike Windows compiler options, which are preceded by a forward slash ("/").

proforder Options

The proforder tool supports the following options:

Tool Option	Default	Description
`-help`		Lists supported options.
`-nologo`		Disables version information. This option is supported on Windows* only.
`-omit_static`		Instructs the tool to omit static functions from function ordering.
`-prof_dir` *dir*		Specifies the directory where the .spi and .dpi file reside.
`-prof_dpi` *file*		Specifies the name of the .dpi file.
`-prof_file` *string*		Selects the .dpi and .spi files that include the substring value in the file name matching the values passed as string.

Tool Option	Default	Description
`-prof_spi` *file*		Specifies the name of the .spi file.
`-o` *file*	`proford.txt`	Specifies an alternate name for the output file.

Using Function Order Lists, Function Grouping, Function Ordering, and Data Ordering Optimizations

Instead of doing a full multi-file interprocedural build of your application by using the compiler option `-ipo` (Linux* OS) or `/Qipo` (Windows* OS), you can obtain some of the benefits by having the compiler and linker work together to make global decisions about where to place the procedures and data in your application. These optimizations are not supported on Mac OS* X systems.

The following table lists each optimization, the type of procedures or global data it applies to, and the operating systems and architectures that it is supported on.

Optimization	Type of Procedure or Data	Supported OS and Architectures
Function Order Lists: Specifies the order in which the linker should link the non-static routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging. Also see Comparison of Function Order Lists and IPO Code Layout.	`EXTERNAL` procedures and library procedures only (not other types of static procedures).	Windows OS: IA-32, Intel® 64, and IA-64 architectures Linux OS: not supported
Function Grouping: Specifies that the linker should place the extern and static routines (procedures) of your program into hot or cold program sections. This optimization can improve	`EXTERNAL` procedures and static procedures only (not library procedures).	Linux OS: IA-32 and Intel 64 architectures Windows OS: not supported

Optimization	Type of Procedure or Data	Supported OS and Architectures
application performance by improving code locality and reduce paging. **Function Ordering:** Enables ordering of static and extern routines using profile information. Specifies the order in which the linker should link the routines (procedures) of your program. This optimization can improve application performance by improving code locality and reduce paging. **Data Ordering:** Enables ordering of static global data items (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) based on profiling information. Specifies the order in which the linker should link global data of your program. This optimization can improve application performance by improving the locality of static global data, reduce paging of large data sets, and improve data cache use.	`EXTERNAL` procedures and static procedures only (not library procedures) Static global data (data in common blocks, module variables, and variables for which the compiler applied the SAVE attribute or statement) only	Linux and Windows OS: IA-32, Intel 64, and IA-64 architectures Linux and Windows OS, IA-32, Intel 64, and IA-64 architectures

You can use only one of the function-related ordering optimizations listed above. However, you can use the Data Ordering optimization with any one of the function-related ordering optimizations listed above, such as Data Ordering with Function Ordering, or Data Ordering with Function Grouping. In this case, specify the `prof-gen` option keyword `globdata` (needed for Data Ordering) instead of `srcpos` (needed for function-related ordering).

The following sections show the commands needed to implement each of these optimizations: function order list, function grouping, function ordering, and data ordering. For all of these optimizations, omit the `-ipo` (Linux* OS) or `/Qipo` (Windows OS) or equivalent compiler option.

Generating a Function Order List (Windows OS)

This section provides an example of the process for generating a function order list. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata`. You would enter commands similar to the following to generate and use a function order list for your Windows application.

1. Compile your program using the `/Qprof-gen:srcpos` option. Use the `/Qprof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Example commands
`ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Example commands
`myprog.exe`

3. Before this step, copy all .dyn and .dpi files into the same directory. Merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file. Use the `/prof_dir` option to specify the directory location of the .dyn files.

Example commands
`profmerge /prof_dir c:\profdata`

4. Generate the function order list using the proforder tool. (By default, the function order list is produced in the file `proford.txt`.)

Example commands
`proforder /prof_dir c:\profdata /o myprog.txt`

5. Compile the application with the generated profile feedback by specifying the `ORDER` option to the linker. Use the `/Qprof-dir` option to specify the directory location of the profile files.

Example commands
`ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /link /ORDER:@MYPROG.txt`

Using Function Grouping (Linux OS)

This section provides a general example of the process for using the function grouping optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `profdata`. You would enter commands similar to the following to use a function grouping for your Linux application.

1. Compile your program using the `-prof-gen` option. Use the `-prof-dir` option to specify the directory location of the profile files. This step creates an instrumented executable.

Example commands
`ifort -o myprog -prof-gen -prof-dir ./profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Example commands
`./myprog`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.

4. Compile the application with the generated profile feedback by specifying the `-prof-func-group` option to request the function grouping as well as the `-prof-use` option to request feedback compilation. Again, use the `-prof-dir` option to specify the location of the profile files.

Example commands
`ifort /exe:myprog file1.f90 file2.f90 -prof-func-group -prof-use -prof-dir ./profdata`

Using Function Ordering

This section provides an example of the process for using the function ordering optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to generate and use function ordering for your application.

1. Compile your program using the `-prof-gen=srcpos` (Linux) or `/Qprof-gen:srcpos` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=srcpos -prof-dir ./profdata file1.f90 file2.f90`
Windows	`ifort /exe:myprog /Qprof-gen:srcpos /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. Change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`

Operating System	Example commands
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-func-order` (Linux) or `/Qprof-func-order` (Windows) option to request the function ordering as well as the `-prof-use` (Linux) or `/Qprof-use` (Windows) option to request feedback compilation. Again, use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-func-order -prof-use`
Windows	`ifort /exe:myprog /Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-func-order /Qprof-use`

Using Data Ordering

This section provides an example of the process for using the data order optimization. Assume you have a Fortran program that consists of the following files: `file1.f90` and `file2.f90`. Additionally, assume you have created a directory for the profile data files called `c:\profdata` (on Windows) or `./profdata` (on Linux). You would enter commands similar to the following to use data ordering for your application.

1. Compile your program using the `-prof-gen=globdata` (Linux) or `/Qprof-gen:globdata` (Windows) option. Use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the directory location of the profile files. This step creates an instrumented executable.

Operating System	Example commands
Linux	`ifort -o myprog -prof-gen=globdata -prof-dir ./profdata file1.f90 file2.f90`
Windows	`ifort /exe:myprog /Qprof-gen:globdata /Qprof-dir c:\profdata file1.f90 file2.f90`

2. Run the instrumented program with one or more sets of input data. If you specified a location other than the current directory, change your directory to the directory where the executables are located. The program produces a `.dyn` file each time it is executed.

Operating System	Example commands
Linux	`./myprog`
Windows	`myprog.exe`

3. Copy all .dyn and .dpi files into the same directory. If needed, you can merge the data from one or more runs of the instrumented program by using the profmerge tool to produce the `pgopti.dpi` file.
4. Compile the application with the generated profile feedback by specifying the `-prof-data-order` (Linux) or `/Qprof-data-order` option to request the data ordering as well as the `-prof-use` (Linux) or `/Qprof-use` (Windows) option to request feedback compilation. Again, use the `-prof-dir` (Linux) or `/Qprof-dir` (Windows) option to specify the location of the profile files.

Operating System	Example commands
Linux	`ifort -o myprog -prof-dir ./profdata file1.f90 file2.f90 -prof-data-order -prof-use`
Windows	`ifort /exe:myprog Qprof-dir c:\profdata file1.f90 file2.f90 /Qprof-data-order /Qprof-use`

Comparison of Function Order Lists and IPO Code Layout

The Intel® compiler provides two methods of optimizing the layout of functions in the executable:

- Using a function order list
- Using the `/Qipo` (Windows) compiler option

Each method has its advantages. A function order list, created with proforder, lets you optimize the layout of non-static functions: that is, external and library functions whose names are exposed to the linker.

The compiler cannot affect the layout order for functions it does not compile, such as library functions. The function layout optimization is performed automatically when IPO is active.

Function Order List Effects

Function Type	IPO Code Layout	Function Ordering with proforder
Static	X	No effect
Extern	X	X
Library	No effect	X

Function Order List Usage Guidelines (Windows*)

Use the following guidelines to create a function order list:

- The order list only affects the order of non-static functions.
-

PGO API Support

API Support Overview

The Profile Information Generation Support (Profile IGS) lets you control the generation of profile information during the instrumented execution phase of profile-guided optimizations.

A set of functions and an environment variable comprise the Profile IGS. The remaining topics in this section describe the associated functions and environment variables.

The compiler sets a `define` for `_PGO_INSTRUMENT` when you compile with either `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*). Without instrumentation, the Profile IGS functions cannot provide PGO API support.

NOTE. The Profile IGS functions are written in C. Fortran applications must call C/C++ functions.

Normally, profile information is generated by an instrumented application when it terminates by calling the standard `exit()` function.

To ensure that profile information is generated, the functions described in this section may be necessary or useful in the following situations:

- The instrumented application exits using a non-standard exit routine.
- The instrumented application is a non-terminating application: `exit()` is never called.
- The application requires control of when the profile information is generated.

You can use the Profile IGS functions in your application by including a header file at the top of any source file where the functions may be used.

Example
`INCLUDE "pgouser.h"`

The Profile IGS Environment Variable

The environment variable for Profile IGS is `INTEL_PROF_DUMP_INTERVAL`. This environment variable may be used to initiate Interval Profile Dumping in an instrumented user application.

See Also

- PGO API Support
- _PGOPTI_Set_Interval_Prof_Dump()

PGO Environment Variables

The environment variables determine the directory in which to store dynamic information files, control the creation of one or multiple dyn files to collect profiling information, and determine whether to overwrite `pgopti.dpi`.

The environment variables are described in the table below.

Variable	Description
INTEL_PROF_DUMP_CUMULATIVE	When using interval profile dumping (initiated by INTEL_PROF_DUMP_INTERVAL or the function _PGOPTI_Set_Interval_Prof_Dump) during the execution of an instrumented user application, allows creation of a single .dyn file to contain profiling information instead of multiple .dyn files. If this environment variable is not set, executing an instrumented user application creates a new .dyn file for each interval. Setting this environment variable is useful for applications that do not terminate or those that terminate abnormally (bypass the normal exit code).
INTEL_PROF_DUMP_INTERVAL	Initiates interval profile dumping in an instrumented user application. This environment variable may be used to initiate Interval Profile Dumping in an instrumented application. See Interval Profile Dumping for more information.
PROF_DIR	Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process.
PROF_DUMP_INTERVAL	Deprecated. Please use INTEL_PROF_DUMP_INTERVAL instead.
PROF_NO_CLOBBER	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges data from all dynamic information files and creates a new pgopti.dpi file if the .dyn files are newer than an existing pgopti.dpi file.

Variable	Description
	When this variable is set the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning. You must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

See the appropriate operating system documentation for instructions on how to specify environment variables and their values.

Dumping Profile Information

The `_PGOPTI_Prof_Dump_All()` function dumps the profile information collected by the instrumented application. The prototype of the function call is listed below.

Syntax

```
void _PGOPTI_Prof_Dump_All(void);
```

An older version of this function, `_PGOPTI_Prof_Dump()`, which will also dump profile information is still available; the older function operates much like `_PGOPTI_Prof_Dump_All()`, except on Linux when used in connection with shared libraries (.so) and `_exit()` to terminate a program. When `_PGOPTI_Prof_Dump_All()` is called before `_exit()` to terminate the program, the new function insures that a .dyn file is created for all shared libraries needing to create a .dyn file. Use `_PGOPTI_Prof_Dump_All()` on Linux to insure portability and correct functionality.

The profile information is generated in a .dyn file (generated in phase 2 of PGO).

Recommended usage

Insert a single call to this function in the body of the function which terminates the user application. Normally, `_PGOPTI_Prof_Dump_All()` should be called just once. It is also possible to use this function in conjunction with `_PGOPTI_Prof_Reset()` function to generate multiple .dyn files (presumably from multiple sets of input data).

Example

```
! Selectively collect profile information
! for the portion of the application
! involved in processing input data.
input_data = get_input_data()
do while (input_data)
  call _PGOPTI_Prof_Reset()
  call process_data(input_data)
  call _PGOPTI_Prof_Dump_All()
  input_data = get_input_data()
end do
end program
```

Dumping Profile Data

This discussion provides an example of how to call the C PGO API routines from Fortran.

As part of the instrumented execution phase of PGO, the instrumented program writes profile data to the dynamic information file (.dyn file).

The profile information file is written after the instrumented program returns normally from `PROGRAM()` or calls the standard exit function.

Programs that do not terminate normally, can use the `_PGOPTI_Prof_Dump_All` function. During the instrumentation compilation, using the `-prof-gen` (Linux* and Mac OS* X) or `/Qprof-gen` (Windows*) option, you can add a call to this function to your program using a strategy similar to the one shown below:

Example

```
interface
  subroutine PGOPTI_Prof_Dump_All()
!DEC$attributes c,alias:'PGOPTI_Prof_Dump_All'::PGOPTI_Prof_Dump_All
  end subroutine
  subroutine PGOPTI_Prof_Reset()
!DEC$attributes c,alias:'PGOPTI_Prof_Reset'::PGOPTI_Prof_Reset
  end subroutine
end interface
call PGOPTI_Prof_Dump_All()
```

CAUTION. You must remove the call or comment it out prior to the feedback compilation with `-prof-use` (Linux and Mac OS X) or `/Qprof-use` (Windows).

Interval Profile Dumping

The `_PGOPTI_Set_Interval_Prof_Dump()` function activates Interval Profile Dumping and sets the approximate frequency at which dumps occur. This function is used in non-terminating applications.

The prototype of the function call is listed below.

Syntax

```
void _PGOPTI_Set_Interval_Prof_Dump(int interval);
```

This function is used in non-terminating applications.

The `interval` parameter specifies the time interval at which profile dumping occurs and is measured in milliseconds. For example, if interval is set to 5000, then a profile dump and reset will occur approximately every 5 seconds. The interval is approximate because the time-check controlling the dump and reset is only performed upon entry to any instrumented function in your application.

Setting the interval to zero or a negative number will disable interval profile dumping, and setting a very small value for the interval may cause the instrumented application to spend nearly all of its time dumping profile information. Be sure to set interval to a large enough value so that the application can perform actual work and substantial profile information is collected.

You can use the profmerge tool to merge the .dyn files.

Recommended usage

Call this function at the start of a non-terminating user application to initiate interval profile dumping. Note that an alternative method of initiating interval profile dumping is by setting the environment variable `INTEL_PROF_DUMP_INTERVAL` to the desired interval value prior to starting the application.

Using interval profile dumping, you should be able to profile a non-terminating application with minimal changes to the application source code.

Resetting the Dynamic Profile Counters

The `_PGOPTI_Prof_Reset()` function resets the dynamic profile counters. The prototype of the function call is listed below.

Syntax
`void _PGOPTI_Prof_Reset(void);`

Recommended usage

Use this function to clear the profile counters prior to collecting profile information on a section of the instrumented application. See the example under Dumping Profile Information.

Dumping and Resetting Profile Information

The `_PGOPTI_Prof_Dump_And_Reset()` function dumps the profile information to a new .dyn file and then resets the dynamic profile counters. Then the execution of the instrumented application continues.

The prototype of the function call is listed below.

Syntax
`void _PGOPTI_Prof_Dump_And_Reset(void);`

This function is used in non-terminating applications and may be called more than once. Each call will dump the profile information to a new .dyn file.

Recommended usage

Periodic calls to this function enables a non-terminating application to generate one or more profile information files (`.dyn` files). These files are merged during the feedback phase (phase 3) of profile-guided optimizations. The direct use of this function enables your application to control precisely when the profile information is generated.

Using High-Level Optimization (HLO) 32

High-Level Optimizations (HLO) Overview

HLO exploits the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization
- Loadpair Optimization
- Predicate Optimization
- Loop Versioning with Runtime Data-Dependence Check (IA-64 architecture only)
- Loop Versioning with Low Trip-Count Check
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests

While the default optimization level, `-O2` (Linux* OS and Mac OS* X) or `/O2` (Windows* OS) option, performs some high-level optimizations (for example, prefetching, complete unrolling, etc.), specifying `-O3` (Linux and Mac OS X) or `/O3` (Windows) provides the best chance for performing loop transformations to optimize memory accesses; the scope of optimizations enabled by these options is different for IA-32 architecture, Intel® 64, and IA-64 architectures.

Applications for the IA-32 and Intel® 64 architectures

In conjunction with the vectorization options, `-ax` and `-x` (Linux and Mac OS X) or `/Qax` and `/Qx` (Windows), the `-O3` (Linux and Mac OS X) or `/O3` (Windows) option causes the compiler to perform more aggressive data dependency analysis than the default `-O2` (Linux and Mac OS X) or `/O2` (Windows).

Compiler prefetching is disabled in favor of the prefetching support available in the processors.

Applications for the IA-32 and IA-64 architectures

The `-O3` (Linux and Mac OS X) or `/O3` (Windows) option enables the `-O2` (Linux and Mac OS X) or `/O2` (Windows) option and adds more aggressive optimizations (like loop transformations); `O3` optimizes for maximum speed, but may not improve performance for some programs.

Applications for the IA-64 architecture

The `-ivdep-parallel` (Linux) or `/Qivdep-parallel` (Windows) option implies there is no loop-carried dependency in the loop where an `IVDEP` directive is specified. (This strategy is useful for sparse matrix applications.)

Tune applications for IA-64 architecture by following these general steps:

1. Compile your program with `-O3` (Linux) or `/O3` (Windows) and `-ipo` (Linux) or `/Qipo` (Windows). Use profile guided optimization whenever possible.
2. Identify hot spots in your code.
3. Generate a high-level optimization report.
4. Check why loops are not software pipelined.
5. Make the changes indicated by the results of the previous steps.
6. Repeat these steps until you achieve the desired performance.

General Application Tuning

In general, you can use the following strategies to tune applications for multiple architectures:

- Use `!DEC$ ivdep` to tell the compiler there is no dependency. You may also need the `-ivdep-parallel` (Linux and Mac OS X) or `/Qivdep-parallel` (Windows) option to indicate there is no loop carried dependency.

- Use `!DEC$ swp` to enable software pipelining (useful for lop-sided control and unknown loop count).
- Use `!DEC$ loop count(n)` when needed.
- If cray pointers are used, use `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) to indicate there is no aliasing.
- Use `!DEC$ distribute point` to split large loops (normally, this is automatically done).
- Check that the prefetch distance is correct. Use `CDEC$ prefetch` to override the distance when it is needed.

Loop Unrolling

The benefits of loop unrolling are as follows:

- Unrolling eliminates branches and some of the code.
- Unrolling enables you to aggressively schedule (or pipeline) the loop to hide latencies if you have enough free registers to keep variables live.
- For processors based on the IA-32 architectures, the processor can correctly predict the exit branch for an inner loop that has 16 or fewer iterations, if that number of iterations is predictable and there are no conditional branches in the loop. Therefore, if the loop body size is not excessive, and the probable number of iterations is known, unroll inner loops for the processors, until they have a maximum of 16 iterations
- A potential limitation is that excessive unrolling, or unrolling of very large loops, can lead to increased code size.

The `-unroll[n]` (Linux* and Mac OS* X) or `/Qunroll:[n]` (Windows*) option controls how the Intel® compiler handles loop unrolling.

Refer to Applying Optimization Strategies for more information.

Linux and Mac OS X	Windows	Description
`-unroll`*n*	`/Qunroll:`*n*	Specifies the maximum number of times you want to unroll a loop. The following examples unrolls a loop four times: `ifort -unroll4 a.f90` (Linux and Mac OS X)

Linux and Mac OS X	Windows	Description
		`ifort /Qunroll:4 a.f90` (Windows) (Linux and Mac OS X) (Windows) **NOTE.** The compilers for IA-64 architecture recognizes only *n* = 0; any other value is ignored. Omitting a value for *n* lets the compiler decide whether to perform unrolling or not. This is the default; the compiler uses default heuristics or defines *n*. Passing 0 as *n* disables loop unrolling; the following examples disables loop unrolling: `ifort -unroll0 a.f90` (Linux and Mac OS X) `ifort /Qunroll:0 a.f90` (Windows) (Linux and Mac OS X) (Windows)

Loop Independence

Loop independence is important since loops that are independent can be parallelized. Independent loops can be parallelized in a number of ways, from the course-grained parallelism of OpenMP*, to fine-grained Instruction Level Parallelism (ILP) of vectorization and software pipelining.

Loops are considered independent when the computation of iteration Y of a loop can be done independently of the computation of iteration X. In other words, if iteration 1 of a loop can be computed and iteration 2 simultaneously could be computed without using any result from iteration 1, then the loops are independent.

Occasionally, you can determine if a loop is independent by comparing results from the output of the loop with results from the same loop written with a decrementing index counter.

For example, the loop shown in example 1 might be independent if the code in example 2 generates the same result.

Example

```
subroutine loop_independent_A (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=0, MAX
    a(j) = b(j)
  end do
end subroutine loop_independent_A
subroutine loop_independent_B (a,b,MAX)
  implicit none
  integer :: j, MAX, a(MAX), b(MAX)
  do j=MAX, 0, -1
    a(j) = b(j)
  end do
end subroutine loop_independent_B
```

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several, general ways.

- Flow Dependency
- Anti Dependency
- Output Dependency
- Reductions

The following sections illustrate the different loop dependencies.

Flow Dependency - Read After Write

Cross-iteration flow dependence is created when variables are written then read in different iterations, as shown in the following example:

Example

```
subroutine flow_dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: A(MAX)
  do j=1, MAX
    A(J)=A(J-1)
  end do
end subroutine flow_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A[1]=A[0];
A[2]=A[1];
```

Recurrence relations feed information forward from one iteration to the next.

Example

```
subroutine time_stepping_loops (a,b,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do j=1, MAX
    a(j) = a(j-1) + b(j)
  end do
end subroutine time_stepping_loops
```

Most recurrences cannot be made fully parallel. Instead, look for a loop further out or further in to parallelize. You might be able to get more performance gains through unrolling.

Anti Dependency - Write After Read

Cross-iteration anti-dependence is created when variables are read then written in different iterations, as shown in the following example:

Example

```
subroutine anti_dependence (A,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX)
  do J=1, MAX
    A(J)=A(J+1)
  end do
end subroutine anti_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A[1]=A[2];
A[2]=A[3];
```

Output Dependency - Write After Write

Cross-iteration output dependence is where variables are written then rewritten in a different iteration. The following example illustrates this type of dependency:

Example

```
subroutine output_dependence (A,B,C,MAX)
  implicit none
  integer :: J,MAX
  real :: a(MAX), b(MAX), c(MAX)
  do J=1, MAX
    A(J)=B(J)
    A(J+1)=C(J)
  end do
end subroutine output_dependence
```

The above example is equivalent to the following lines for the first few iterations:

Sample Iterations

```
A[1]=B[1];
A[2]=C[1];
A[2]=B[2];
A[3]=C[2];
```

Reductions

The Intel® compiler can successfully vectorize or software pipeline (SWP) most loops containing reductions on simple math operators like multiplication (*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

Example

```
subroutine reduction (sum,c,MAX)
  implicit none
  integer :: j,MAX
  real :: sum, c(MAX)
  do J=0, MAX
    sum = sum + c(j)
  end do
end subroutine reduction
```

The compiler might occasionally misidentify a reduction and report flow-, anti-, output-dependencies or sometimes loop-carried memory-dependency-edges; in such cases, the compiler will not vectorize or SWP the loop.

Prefetching with Options

The goal of prefetch insertion optimization is to reduce cache misses by providing hints to the processor about when data should be loaded into the cache. The prefetch optimization is enabled or disabled by the `-opt-prefetch` (Linux* and Mac OS* X) or `/Qopt-prefetch` (Windows*) compiler option. This option also allows you to specify the level of software prefetching.

To facilitate compiler optimization:

- Minimize use of global variables and pointers.
- Minimize use of complex control flow.
- Choose data types carefully and avoid type casting.

In addition to the `-opt-prefetch` (Linux and Mac OS X) or `/Qopt-prefetch` (Windows) option, an intrinsic subroutine `mm_prefetch` and compiler directive `prefetch` are also available.

The architecture affects the option behavior. For more information about the differences, see the following topic:

- `-opt-prefetch` compiler option

See Also

- Using High-Level Optimization (HLO)
- Other Resources

Optimization Support Features 33

Optimization Support Features Overview

This section describes the language extensions that enable you to optimize your source code directly.

- Loop Support
- Loop Unrolling Support
- Vectorization Support
- Prefetch Support
- Software Pipelining (IA-64 Architecture)

See General Compiler Directives for more information about these directives.

Loop Support

LOOP COUNT Directive

The `LOOP COUNT` directive can affect heuristics used in vectorization, loop-transformations, and software pipelining (IA-64 architecture). The directive can specific the minimum, maximum, or average number of iterations for a `DO` loop. In addition, a list of commonly occurring values can be specified. This behavior might help the compiler generate multiple versions and perform complete unrolling.

The syntax for this directive supports the following syntax variations:

Syntax
!DEC$ LOOP COUNT (*n*)
!DEC$ LOOP COUNT = *n*
or
!DEC$ LOOP COUNT (*n1*[,*n2*]...)
!DEC$ LOOP COUNT = *n1*[,*n2*]...
or
!DEC$ LOOP COUNT MAX(*n*), AVG(*n*)
!DEC$ LOOP COUNT MAX=*n*, MIN=*n*, AVG=*n*

The syntax variations support the following arguments:

Argument	Description
(*n*) or =*n*	Non-negative integer value. The compiler will attempt to iterate the next loop the number of times specified in *n*; however, the number of iterations is not guaranteed.
(*n1*[,*n2*]...) or =*n1*[,*n2*]...	Non-negative integer values. The compiler will attempt to iterate the next loop the number of time specified by *n1* or *n2*, or some other unspecified number of times. This behavior allows the compiler some flexibility in attempting to unroll the loop. The number of iterations is not guaranteed.
min(*n*), max(*n*), >avg(*n*) or min=*n*, max=*n*, avg=*n*	Non-negative integer values. Specify one or more in any order without duplication. The compiler insures the next loop iterates for the specified maximum, minimum, or average number (*n1*) of times. The specified number of iterations is guaranteed for min and max.

You can specify more than one directive for a single loop; however, do not duplicate the directive.

The following example illustrates how to use the directive to iterate through the loop and enable software pipelining on IA-64 architectures.

Using Loop Count (n)

```
subroutine loop_count(a, b, N)
integer :: i, N, b(N), c(N)
!DEC$ LOOP COUNT (1000)
 do i = 1, 1000
   b(i) = a(i) + 1
 enddo
end subroutine loop_count
```

The following example illustrates how to use the directive to iterate through the loop a minimum of three, a maximum of ten, and average of five times.

Using Loop Count min, max, avg

```
!DEC$ LOOP COUNT MIN(3), MAX(10), AVG(5)
DO i = 1, 15
  PRINT i
END DO
```

DISTRIBUTE POINT Directive

The `DISTRIBUTE POINT` directive instructs the compiler to prefer loop distribution at the location indicated. The syntax for this directive is shown below:

Syntax

```
!DEC$ DISTRIBUTE POINT
```

Loop distribution can cause large loops to be distributed into smaller ones. This strategy can enable software pipelining for the new, smaller loops (IA-64 architecture).

- If the directive is placed inside a loop, distribution is performed after the directive and any loop-carried dependency is ignored.

- If the directive is placed before a loop, the compiler determines where to distribute the loops and observes data dependency. If they are placed inside the loop, the compiler supports multiple instances of the directive.

Using distribute point

```
subroutine dist1(a, b, c, d, N)
  integer :: i, N, a(N), b(N), c(N), d(N)
 !DEC$ DISTRIBUTE POINT
  do i = 1, N
    b(i) = a(i) + 1
    c(i) = a(i) + b(i)
 ! Compiler will decide where to distribute.
 ! Data dependency is observed.
    d(i) = c(i) + 1
  enddo
end subroutine dist1
subroutine dist2(a, b, c, d, N)
  integer :: i, N, a(N), b(N), c(N), d(N)
  do i = 1, N
    b(i) = a(i) + 1
    !DEC$

 DISTRIBUTE POINT
 ! Distribution will start here, ignoring all
 ! loop-carried dependency.
    c(i) = a(i) + b(i)
    d(i) = c(i) + 1
  enddo
end subroutine dist2
```

Loop Unrolling Support

The UNROLL[*n*] directive tells the compiler how many times to unroll a counted loop. The general syntax for this directive is shown below:

Syntax

```
!DEC$ UNROLL(n)
```

where *n* is an integer constant from 0 through 255.

The UNROLL directive must precede the DO statement for each DO loop it affects. If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted or if it is outside the allowed range, the optimizer assigns the number of times to unroll the loop.

This directive is supported only when option -O3 (Linux* OS and Mac OS* X) or /O3 (Windows* OS) is used. The UNROLL directive overrides any setting of loop unrolling from the command line.

Currently, the directive can be applied only for the innermost loop nest. If applied to the outer loop nests, it is ignored. The compiler generates correct code by comparing *n* and the loop count.

Example

```
subroutine unroll(a, b, c, d)
  integer :: i, b(100), c(100), d(100)
  !DEC$ UNROLL(4)
  do i = 1, 100
    b(i) = a(i) + 1
    d(i) = c(i) + 1
  enddo
end subroutine unroll
```

Vectorization Support

IVDEP Directive

The `IVDEP` directive instructs the compiler to ignore assumed vector dependences. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This directive overrides that decision. Use `IVDEP` only when you know that the assumed loop dependences are safe to ignore.

For example, if the expression `j >= 0` is always true in the code fragment bellow, the `IVDEP` directive can communicate this information to the compiler. This directive informs the compiler that the conservatively assumed loop-carried flow dependences for values `j < 0` can be safely ignored:

Example

```
!DEC$ IVDEP
  do i = 1, 100
     a(i) = a(i+j)
  enddo
```

NOTE. The proven dependences that prevent vectorization are not ignored, only assumed dependences are ignored.

The usage of the directive differs depending on the loop form, see examples below.

Example: Loop 1

```
do i
     = a(*) + 1
    a(*) =
 enddo
```

Example: Loop 2

```
do i
    a(*) =
         = a(*) + 1
 enddo
```

For loops of the form 1, use old values of *a*, and assume that there is no loop-carried flow dependencies from `DEF` to `USE`.

For loops of the form 2, use new values of *a*, and assume that there is no loop-carried anti-dependencies from `USE` to `DEF`.

In both cases, it is valid to distribute the loop, and there is no loop-carried output dependency.

Example 1

```
!DEC$ IVDEP
do j=1,n
  a(j)= a(j+m) + 1
enddo
```

Example 2

```
!DEC$ IVDEP
do j=1,n
  a(j) = b(j) + 1
  b(j) = a(j+m) + 1
enddo
```

Example 1 ignores the possible backward dependencies and enables the loop to get software pipelined.

Example 2 shows possible forward and backward dependencies involving array *a* in this loop and creating a dependency cycle. With `IVDEP`, the backward dependencies are ignored.

IVDEP has options: `IVDEP:LOOP` and `IVDEP:BACK`. The `IVDEP:LOOP` option implies no loop-carried dependencies. The `IVDEP:BACK` option implies no backward dependencies.

The IVDEP directive is also used for IA-64 architecture based applications.

Overriding the Efficiency Heuristics in the Vectorizer

In addition to IVDEP directive, there are VECTOR directives that can be used to override the efficiency heuristics of the vectorizer:

- `VECTOR ALWAYS`
- `NOVECTOR`
- `VECTOR ALIGNED`
- `VECTOR UNALIGNED`

- VECTOR NONTEMPORAL

The VECTOR directives control the vectorization of the subsequent loop in the program, but the compiler does not apply them to nested loops. Each nested loop needs its own directive preceding it. You must place the vector directive before the loop control statement.

VECTOR ALWAYS and NOVECTOR Directives

The VECTOR ALWAYS directive overrides the efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized, that is: use IVDEP to ignore assumed dependences.

The VECTOR ALWAYS directive can be used to override the default behavior of the compiler in the following situation. Vectorization of non-unit stride references usually does not exhibit any speedup, so the compiler defaults to not vectorizing loops that have a large number of non-unit stride references (compared to the number of unit stride references). The following loop has two references with stride 2.

Vectorization would be disabled by default, but the directive overrides this behavior.

Example

```
!DEC$ VECTOR ALWAYS
do i = 1, 100, 2
  a(i) = b(i)
enddo
```

If, on the other hand, avoiding vectorization of a loop is desirable (if vectorization results in a performance regression rather than improvement), the NOVECTOR directive can be used in the source text to disable vectorization of a loop. For instance, the compiler vectorizes the following example loop by default. If this behavior is not appropriate, the NOVECTOR directive can be used, as shown below.

Example

```
!DEC$ NOVECTOR
do i = 1, 100
  a(i) = b(i) + c(i)
enddo
```

VECTOR ALIGNED/UNALIGNED Directives

Like VECTOR ALWAYS, these directives also override the efficiency heuristics. The difference is that the qualifiers UNALIGNED and ALIGNED instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

NOTE. The directives VECTOR [ALWAYS|UNALIGNED|ALIGNED] should be used with care. Overriding the efficiency heuristics of the compiler should only be done if the programmer is absolutely sure the vectorization will improve performance. Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a run-time exception in case some of the access patterns are actually unaligned.

The VECTOR NONTEMPORAL Directive

The VECTOR NONTEMPORAL directive results in streaming stores on the systems based on IA-32 and Intel® 64 architectures. A floating-point type loop together with the generated assembly are shown in the example below. For large *n*, significant performance improvements result on a Pentium 4 system over a non-streaming implementation.

The following example shows the VECTOR NONTEMPORAL directive that directs the compiler to use streaming store on the array *a* but not for the array *b*:

Example

```
subroutine set(a,b,n)
integer i,n
real a(n), b(n)
!DEC$ VECTOR NONTEMPORAL (a)
!DEC$ VECTOR ALIGNED
do i = 1, n
  a(i) = 1
  b(i) = 1
enddo
end
program setit
parameter(n=1024*1204)
real a(n), b(n)
integer i
do i = 1, n
  a(i) = 0
enddo
call set(a,b,n)
do i = 1, n
  if (a(i)+b(i).ne.2) then
    print *, 'failed nontemp.f', a(i), i
    stop
  endif
enddo
print *, 'passed nontemp.f'
```

Example
`end`

Prefetching Support

Data prefetching refers to loading data from a relatively slow memory into a relatively fast cache before the data is needed by the application. Data prefetch behavior depends on the architecture:

- IA-64 architecture: The Intel® compiler generally issues prefetch instructions when you specify `-O1`, `-O2`, and `-O3` (Linux*) or `/O1`, `/O2`, and `/O3` (Windows*).
- IA-32 and Intel® 64 architectures: The processor identifies simple, regular data access patterns and performs a hardware prefetch. The compiler will only issue prefetch instructions for more complicated data access patterns where a hardware prefetch is not expected.

Issuing prefetches improves performance in most cases; however, there are cases where issuing prefetch instructions might slow application performance. Experiment with prefetching; it can be helpful to turn prefetching on or off with a compiler option while leaving all other optimizations unaffected to isolate a suspected prefetch performance issue. See Prefetching with Options for information on using compiler options for prefetching data.

There are two primary methods of issuing prefetch instructions. One is by using compiler directives and the other is by using compiler intrinsics.

PREFETCH and NOPREFETCH Directives

The `PREFETCH` and `NOPREFETCH` directives are supported by Itanium® processors only. These directives assert that the data prefetches be generated or not generated for some memory references. This affects the heuristics used in the compiler.

If loop includes expression `A(j)`, placing `PREFETCH A` in front of the loop, instructs the compiler to insert prefetches for `A(j + d)` within the loop. `d` is the number of iterations ahead to prefetch the data and is determined by the compiler. This directive is supported with optimization levels of `-O1` (Linux*) or `/O1` (Windows*) or higher. Remember that `-O2` or `/O2` is the default optimization level.

Example

```
!DEC$ NOPREFETCH c
!DEC$ PREFETCH a
  do i = 1, m
    b(i) = a(c(i)) + 1
  enddo
```

The following example is for IA-64 architecture only:

Example

```
do j=1,lastrow-firstrow+1
i = rowstr(j)
iresidue = mod( rowstr(j+1)-i, 8 )
sum = 0.d0
!DEC$ NOPREFETCH a,p,colidx
do k=i,i+iresidue-1
  sum = sum +  a(k)*p(colidx(k))
enddo
!DEC$ NOPREFETCH colidx
!DEC$ PREFETCH a:1:40
!DEC$ PREFETCH p:1:20
do k=i+iresidue, rowstr(j+1)-8, 8
  sum = sum + a(k  )*p(colidx(k  ))
  &      + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
  &      + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
  &      + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
  &      + a(k+7)*p(colidx(k+7))
enddo
q(j) = sum
enddo
```

Intrinsics

Before inserting compiler intrinsics, experiment with all other supported compiler options and directives. Compiler intrinsics are less portable and less flexible than either a compiler option or compiler directives.

Directives enable compiler optimizations while intrinsics perform optimizations. As a result, programs with directives are more portable, because the compiler can adapt to different processors, while the programs with intrinsics may have to be rewritten/ported for different processors. This is because intrinsics are closer to assembly programming.

The compiler supports an intrinsic subroutine `mm_prefetch`. In contrast the way the `prefetch` directive enables a data prefetch from memory, the subroutine `mm_prefetch` prefetches data from the specified address on one memory cache line. The `mm_prefetch` subroutine is described in the *Intel® Fortran Language Reference*.

Software Pipelining Support (IA-64 Architecture)

The `SWP` directive indicates preference for loops to be software pipelined. The directive does not help data dependency, but overrides heuristics based on profile counts or unequal control flow.

The syntax for this directive is shown below:

Syntax
`!DEC$ SWP`

The Software Pipelining optimization triggered by the `SWP` directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages, allowing increased instruction level parallelism.

This strategy can reduce the impact of long-latency operations, resulting in faster loop execution. Loops chosen for software pipelining are always innermost loops that do not contain procedure calls that are not inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see loop unrolling options).

You can view an optimization report to see whether software pipelining was applied (see Optimizer Report Generation).

The following example demonstrates on way of using the directive to instruct the compiler to attempt software pipelining.

Example: Using SWP

```
subroutine swp(a, b)
  integer :: i, a(100), b(100)
  !DEC$ SWP
  do i = 1, m
    if (a(i) .eq. 0)

 then
       b(i) = a(i) + 1
    else
       b(i) = a(i)/c(i)
    endif
  enddo
end subroutine swp
```

About Register Allocation

The Intel® Compiler for IA-32 and Intel® 64 architectures contains an advanced, region-based register allocator. Register allocation can be influenced using the `-opt-ra-region-strategy` (Linux* and Mac OS* X) and `/Qopt-ra-region-strategy` (Windows*) option.

The register allocation high-level strategy when compiling a routine is to partition the routine into regions, assign variables to registers or memory within each region, and resolve discrepancies at region boundaries. The overall quality of the allocation depends heavily on the region partitioning.

By default, the Intel Compiler selects the best region partitioning strategy, but the `-opt-ra-region-strategy` (Linux* and Mac OS X) and `/Qopt-ra-region-strategy` (Windows) option allows you to experiment with the other available allocation strategies, which might result in better performance in some cases. The option provides several different arguments that allow you to specify the following allocation strategies:

- routine = a region for each routine

- trace = a region for each trace
- loop = a region for each loop
- block = a region for each block
- default = the compiler selects best allocation strategy

See the `/Qopt-ra-region-strategy` (Windows) `-opt-ra-region-strategy` (Linux and Mac OS X) compiler option for additional information.

The option can affect compile time. Register allocation is a relatively costly operation, and the time spent in register allocation tends to grow as the number of regions increases. Expect relative compile time to increase in the order listed, from shortest to longest:

1. routine-based regions (shortest)
2. loop-based regions
3. trace-based regions
4. block-based regions (longest)

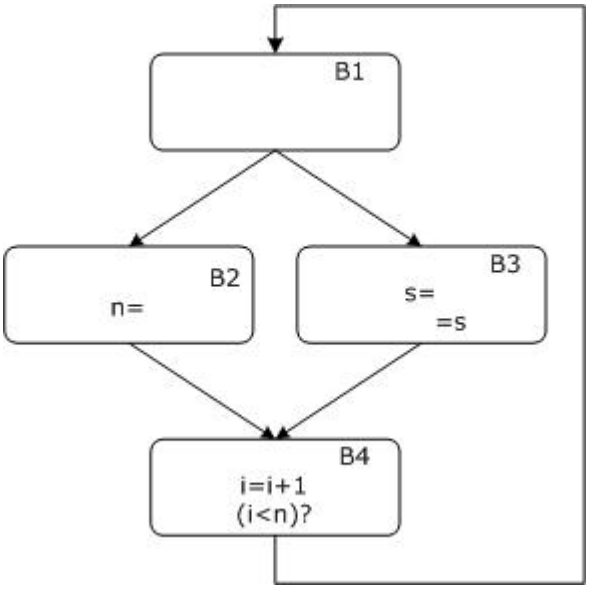
Trace-based regions tend to work very well when profile guided optimizations are enabled. The allocator is able to construct traces that accurately reflect the hot paths through the routine.

In the absence of profile information, loop-based regions tend to work well because the execution profile of a program tends to match its loop structure. In programs where the execution profile does not match the loop structure, routine- or block-based regions can produce better allocations.

Block-based regions provide maximum flexibility to the allocator and in many cases can produce the best allocation; however, the allocator is sometimes over-aggressive with block-based regions about allocating variables to registers; the behavior can lead to poor allocations in the IA-32 and Intel® 64 architectures where registers can be scarce resources.

Register Allocation Example Scenarios

Consider the following example, which illustrates a control flow that results from a simple if-then-else statement within a loop.



There are 3 variables in this loop: *i*, *s*, and *n*. For this example, assume there are only two registers available to hold these three variables; one, or more, variable will need to be stored in memory for at least part of the loop.

The best choice depends on which path through the loop is more frequently executed. For example, if B1, B2, B4 is the hot path, keeping variables *i* and *n* in registers along that path and saving and restoring one of them in B3 to free up a register for *s* is the best strategy. That scenario avoids all memory accesses on the hot path. If B1, B3, B4 is the hot path, the best strategy is to keep variables *i* and *s* in registers and to store n in memory, since there are no assignments to *n* along the path. This strategy results in a single memory read on the hot path. If both paths are executed with equal frequency, the best strategy is to save and restore either *i* or *n* around B3 just like in the B1, B2, B4 case. That case avoids all memory accesses on one path and results in a single memory write and a single memory read on the other path.

The compiler generates two significantly different allocations for this example depending on the region strategy; the preferred result depends on the runtime behavior of the program, which may not be known at compile time.

With a routine- or a loop-based strategy, all four blocks will be allocated together. The compiler picks a variable to store in memory based on expected costs. In this example, the allocator will probably select variable *n*, resulting in a memory write in B2 and a memory read in B3.

With a trace-based strategy, the compiler uses estimates of execution frequency to select the most frequently executed path through the loop. When profile guided optimizations are enabled the estimates are based on the concrete information about the runtime behavior of the instrumented program. If the PGO information accurately reflects typical application behavior, the compiler produces highly accurate traces. In other cases, the traces are not necessarily an accurate reflection of the hot paths through the code.

Suppose in this example that the compiler selects B1, B2, B4 path as the hot trace. The compiler will assign these three blocks to one region, and B3 will be in a separate region. There are only two variables in the larger region, so both may be kept in registers. In the region containing just B3 either `i` or `n` are stored in memory, and the compiler makes an arbitrary choice between the two variables. In a block-based strategy, each block is a region. In the B1, B2, B4 path there are sufficient registers to hold the two variables: `i` and `n`. The region containing B3 is treated just like the trace-based case; either `i` or `n` will be stored in memory.

Programming Guidelines 34

Coding Guidelines for Intel® Architectures

This topic provides general guidelines for coding practices and techniques for using:

- IA-32 and Intel® 64 architectures supporting MMX™ technology and Intel® Streaming SIMD Extensions Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Streaming SIMD Extensions 4 (SSE4)
- IA-64 architecture

This section describes practices, tools, coding rules and recommendations associated with the architecture features that can improve the performance for processors based on IA-32, Intel® 64, and IA-64 architectures.

NOTE. If a guideline refers to a particular architecture only, this architecture is explicitly named. The default is for IA-32 architectures.

Performance of compiler-generated code may vary from one compiler to another. Intel® Fortran Compiler generates code that is optimized for Intel architectures. You can significantly improve performance by using various compiler optimization options. In addition, you can help the compiler to optimize your Fortran program by following the guidelines described in this section.

To achieve optimum processor performance in your Fortran application, do the following:

- avoiding memory access stalls
- ensuring good floating-point performance
- ensuring good SIMD integer performance
- using vectorization

The following sections summarize and describe coding practices, rules and recommendations associated with the features that will contribute to optimizing the performance on Intel architecture-based processors.

Memory Access

The Intel compiler lays out arrays in column-major order. For example, in a two-dimensional array, elements `A(22,34)` and `A(23,34)` are contiguous in memory. For best performance, code arrays so that inner loops access them in a contiguous manner.

Consider the following examples. The code in example 1 will likely have higher performance than the code in example 2.

Example 1

```
subroutine contiguous(a, b, N)
  integer :: i, j, N, a(N,N), b(N,N)
  do j = 1, N
    do i = 1, N
      b(i, j) = a(i, j) + 1
    end do
  end do
end subroutine contiguous
```

The code above illustrates access to arrays `A` and `B` in the inner loop `I` in a contiguous manner which results in good performance; however, the following example illustrates access to arrays `A` and `B` in inner loop `J` in a non-contiguous manner which results in poor performance.

Example 2

```
subroutine non_contiguous(a, b, N)
  integer :: i, j, N, a(N,N), b(N,N)
  do i = 1, N
    do j = 1, N
      b(i, j) = a(i, j) + 1
    end do
  end do
end subroutine non_contiguous
```

The compiler can transform the code so that inner loops access memory in a contiguous manner. To do that, you need to use advanced optimization options, such as `-O3` (Linux* OS) or `/O3` (Windows* OS) for IA-32, Intel® 64, and IA-64 architectures, and `-O3` (Linux) or `/O3` (Windows) and `-ax` (Linux) or `/Qax` (Windows) for IA-32 architecture only.

Memory Layout

Alignment is an increasingly important factor in ensuring good performance. Aligned memory accesses are faster than unaligned accesses. If you use the interprocedural optimization on multiple files, the `-ipo` (Linux) or `/Qipo` (Windows) option, the compiler analyzes the code and decides whether it is beneficial to pad arrays so that they start from an aligned boundary. Multiple arrays specified in a single common block can impose extra constraints on the compiler.

For example, consider the following `COMMON` statement:

Example 3

```
COMMON /AREA1/ A(200), X, B(200)
```

If the compiler added padding to align `A(1)` at a 16-byte aligned address, the element `B(1)` would not be at a 16-byte aligned address. So it is better to split `AREA1` as follows.

Example 4

```
COMMON /AREA1/ A(200)
COMMON /AREA2/ X
COMMON /AREA3/ B(200)
```

The above code provides the compiler maximum flexibility in determining the padding required for both `A` and `B`.

Setting Data Type and Alignment

Alignment of data affects these kinds of variables:

- Those that are dynamically allocated. (Dynamically allocated data allocated with `ALLOCATE` is 8-byte aligned.)
- Those that are members of a data structure
- Those that are global or local variables

- Those that are parameters passed on the stack

For best performance, align data as follows:

- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned four byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.
- Align 128-bit data so that its base address is a multiple of sixteen (8-byte boundaries).

Causes of Unaligned Data and Ensuring Natural Alignment

For optimal performance, make sure your data is aligned naturally. A natural boundary is a memory address that is a multiple of the data item's size. For example, a `REAL (KIND=8)` data item aligned on natural boundaries has an address that is a multiple of 8. An array is aligned on natural boundaries if all of its elements are so aligned.

All data items whose starting address is on a natural boundary are naturally aligned. Data not aligned on a natural boundary is called unaligned data.

Although the Intel® compiler naturally aligns individual data items when it can, certain Fortran statements can cause data items to become unaligned.

You can use the align command-option to ensure naturally aligned data, but you should check and consider reordering data declarations of data items within common blocks, derived-type structures, and record structures as follows:

- Carefully specify the order and sizes of data declarations to ensure naturally aligned data.
- Start with the largest size numeric items first, followed by smaller size numeric items, and then non-numeric (character) data.

The `!DEC$ ATTRIBUTES ALIGN` directive specifies the byte alignment for a variable.

Common blocks (`COMMON` statement), derived-type data, and record structures (`RECORD` statement) usually contain multiple items within the context of the larger structure.

The following statements can cause unaligned data:

Statement	Options	Description
Common blocks (COMMON statement)	`commons` or `dcommons`	The order of variables in the `COMMON` statement determines their storage

Statement	Options	Description
		order. Unless you are sure that the data items in the common block will be naturally aligned, specify either `-align commons` or `-align dcommons` (Linux*) or `/align:commons` or `/align:dcommons` (Windows*), depending on the largest data size used.
Derived-type (user-defined) data	`records` or `sequence`	Derived-type data items are declared after a `TYPE` statement. If your data includes derived-type data structures, you should use the `-align records` (Linux) or `/align:records` (Windows) option, unless you are sure that the data items in the derived-type structures will be naturally aligned. If you omit the sequence statement, the `-align records` (Linux) or `/align:records` (Windows) option (default) ensures all data items are naturally aligned. If you specify the `SEQUENCE` statement, the `-align records` (Linux) or `/align:records` (Windows) option is prevented from adding necessary padding to

Statement	Options	Description
		avoid unaligned data (data items are packed) unless you specify `SEQUENCE`. When you use `SEQUENCE`, you should specify data declaration order so that all data items are naturally aligned.
Record structures (RECORD and STRUCTURE statements)	`record` or `structure`	Intel Fortran record structures usually contain multiple data items. The order of variables in the `STRUCTURE` statement determines their storage order. The `RECORD` statement names the record structure. Record structures are an Intel Fortran language extension. If your data includes record structures, you should use the `-align records` (Linux) or `/align:records` (Windows) option, unless you are sure that the data items in the record structures will be naturally aligned.
EQUIVALENCE statements		`EQUIVALENCE` statements can force unaligned data or cause data to span natural boundaries. For more information, see the *Intel® Fortran Language Reference*.

To avoid unaligned data in a common block, derived-type data, or record structure (extension), use one or both of the following:

- For new programs or for programs where the source code declarations can be modified easily, plan the order of data declarations with care. For example, you should order variables in a COMMON statement such that numeric data is arranged from largest to smallest, followed by any character data (see the data declaration rules in Ordering Data Declarations to Avoid Unaligned Data below.
- For existing programs where source code changes are not easily done or for array elements containing derived-type or record structures, you can use command line options to request that the compiler align numeric data by adding padding spaces where needed.

Other possible causes of unaligned data include unaligned actual arguments and arrays that contain a derived-type structure or Intel Fortran record structure as detailed below.

- When actual arguments from outside the program unit are not naturally aligned, unaligned data access occurs. Intel Fortran assumes all passed arguments are naturally aligned and has no information at compile time about data that will be introduced by actual arguments during program execution.
- For arrays where each array element contains a derived-type structure or Intel Fortran record structure, the size of the array elements may cause some elements (but not the first) to start on an unaligned boundary.
- Even if the data items are naturally aligned within a derived-type structure without the SEQUENCE statement or a record structure, the size of an array element might require use of the `-align records` (Linux) or `/align:records` (Windows) option to supply needed padding to avoid some array elements being unaligned.
- If you specify `-align norecords` (Linux) or `/align:norecords` (Windows) or specify `-vms` (Linux) or `/Qvms` (Windows) without RECORDS no padding bytes are added between array elements. If array elements each contain a derived-type structure with the SEQUENCE statement, array elements are packed without padding bytes regardless of the Fortran command options specified. In this case, some elements will be unaligned.
- When the `-align records` (Linux) or `/align:records` (Windows)option is in effect, the number of padding bytes added by the compiler for each array element is dependent on the size of the largest data item within the structure. The compiler determines the size of the array elements as an exact multiple of the largest data item in the derived-type structure without the SEQUENCE statement or a record structure. The compiler then adds the appropriate number of padding bytes. For instance, if a structure contains an 8-byte floating-point number followed by a 3-byte character variable, each element contains five bytes of padding (16 is an exact multiple of 8). However, if the structure contains one 4-byte floating-point number, one 4-byte integer, followed by a 3-byte character variable, each element would contain one byte of padding (12 is an exact multiple of 4).

Checking for Inefficient Unaligned Data

During compilation, the Intel compiler naturally aligns as much data as possible. Exceptions that can result in unaligned data are described above.

Because unaligned data can slow run-time performance, it is worthwhile to:

- Double-check data declarations within common block, derived-type data, or record structures to ensure all data items are naturally aligned (see the data declaration rules in the subsection below). Using modules to contain data declarations can ensure consistent alignment and use of such data.
- Avoid the `EQUIVALENCE` statement or use it in a manner that cannot cause unaligned data or data spanning natural boundaries.
- Ensure that passed arguments from outside the program unit are naturally aligned.
- Check that the size of array elements containing at least one derived-type data or record structure (extension) cause array elements to start on aligned boundaries (see the previous subsection).
- There are two ways unaligned data might be reported:
  - During compilation, warning messages are issued for any data items that are known to be unaligned (unless you specify the `-warn noalignments` (or `-warn none`) (Linux) or `/warn:noalignments` (or `/warn:none`) (Windows) option that suppresses all warnings).
  - During program execution, warning messages are issued for any data that is detected as unaligned. The message includes the address of the unaligned access.

Consider the following run-time message:

Example

```
Unaligned access pid=24821 <a.out> va=140000154, pc=3ff80805d60,
ra=1200017bc
```

This message shows that:

- The statement accessing the unaligned data (program counter) is located at 3ff80805d60
- The unaligned data is located at address 140000154

Ordering Data Declarations to Avoid Unaligned Data

For new programs or when the source declarations of an existing program can be easily modified, plan the order of your data declarations carefully to ensure the data items in a common block, derived-type data, record structure, or data items made equivalent by an `EQUIVALENCE` statement will be naturally aligned.

Use the following rules to prevent unaligned data:

- Always define the largest size numeric data items first.
- If your data includes a mixture of character and numeric data, place the numeric data first.
- Add small data items of the correct size (or padding) before otherwise unaligned data to ensure natural alignment for the data that follows.

When declaring data, consider using explicit length declarations, such as specifying a `KIND` parameter. For example, specify `INTEGER(KIND=4)` (or `INTEGER(4)`) rather than `INTEGER`. If you do use a default size (such as `INTEGER`, `LOGICAL`, `COMPLEX`, and REAL), be aware that the following compiler options can change the size of an individual field's data declaration size and thus can alter the data alignment of a carefully planned order of data declarations:

Operating System	Compiler Option
Windows*	`/4I` or `/4R`
Linux* and Mac OS* X	`-integer_size` or `-real_size`

Using the suggested data declaration guidelines minimizes the need to use the `-align keyword` (Linux) or `/align:keyword` (Windows) options to add padding bytes to ensure naturally aligned data. In cases where the `-align keyword` (Linux) or `/align:keyword` (Windows) options are still needed, using the suggested data declaration guidelines can minimize the number of padding bytes added by the compiler.

Arranging Data Items in Common Blocks

The order of data items in a COMMON statement determines the order in which the data items are stored. Consider the following declaration of a common block named X:

Example

```
logical (kind=2) flag
integer          iarry_i(3)
character(len=5) name_ch
common /x/ flag, iarry_i(3), name_ch
```

As shown in Figure 1-1, if you omit the appropriate Fortran command options, the common block will contain unaligned data items beginning at the first array element of IARRY_I.

Figure 22: Figure 1-1 Common Block with Unaligned Data

As shown in Figure 1-2, if you compile the program units that use the common block with the -align commons (Linux) or /align:commons (Windows) option, data items will be naturally aligned.

Figure 23: Figure 1-2 Common Block with Naturally Aligned Data

Because the common block X contains data items whose size is 32 bits or smaller, specify the -align commons (Linux) or /align:commons (Windows) option. If the common block contains data items whose size might be larger than 32 bits (such as REAL (KIND=8) data), use the -align dcommons (Linux) or /align:dcommons (Windows) option.

If you can easily modify the source files that use the common block data, define the numeric variables in the COMMON statement in descending order of size and place the character variable last. This provides more portability, ensures natural alignment without padding, and does not require the Fortran command options -align commons (Linux) or -align dcommons (Linux) or /align commons (Windows) or /align:dcommons (Windows):

Example

```
LOGICAL (KIND=2) FLAG
INTEGER          IARRY_I(3)
CHARACTER(LEN=5) NAME_CH
COMMON /X/ IARRY_I(3), FLAG, NAME_CH
```

As shown in Figure 1-3, if you arrange the order of variables from largest to smallest size and place character data last, the data items will be naturally aligned.

Figure 24: Figure 1-3 Common Block with Naturally Aligned Reordered Data

When modifying or creating all source files that use common block data, consider placing the common block data declarations in a module so the declarations are consistent. If the common block is not needed for compatibility (such as file storage or Fortran 77 use), you can place the data declarations in a module without using a COMMON block.

Arranging Data Items in Derived-Type Data

Like common blocks, derived-type data may contain multiple data items (members).

Data item components within derived-type data will be naturally aligned on up to 64-bit boundaries, with certain exceptions related to the use of the SEQUENCE statement and Fortran options.

Intel Fortran stores a derived data type as a linear sequence of values, as follows:

- If you specify the SEQUENCE statement, the first data item is in the first storage location and the last data item is in the last storage location. The data items appear in the order in which they are declared. The Fortran options have no effect on unaligned data, so data

declarations must be carefully specified to naturally align data. The `-align sequence` (Linux) or `/align:sequence` (Windows) option specifically aligns data items in a `SEQUENCE` derived-type on natural boundaries.

- If you omit the `SEQUENCE` statement, the Intel Fortran adds the padding bytes needed to naturally align data item components, unless you specify the `-align norecords` (Linux) or `/align:norecords` (Windows) option.

Consider the following declaration of array `CATALOG_SPRING` of derived-type `PART_DT`:

Example

```
MODULE DATA_DEFS
  TYPE PART_DT
    INTEGER           IDENTIFIER
    REAL              WEIGHT
    CHARACTER(LEN=15) DESCRIPTION
  END TYPE PART_DT
  TYPE (PART_DT) CATALOG_SPRING(30)
...
END MODULE DATA_DEFS
```

As shown in Figure 1-4, the largest numeric data items are defined first and the character data type is defined last. There are no padding characters between data items and all items are naturally aligned. The trailing padding byte is needed because `CATALOG_SPRING` is an array; it is inserted by the compiler when the `-align records` (Linux) or `/align:records` (Windows) option is in effect.

Figure 25: Figure 1-4 Derived-Type Naturally Aligned Data (in CATALOG_SPRING : (,))

Arranging Data Items in Intel Fortran Record Structures

Intel Fortran supports record structures provided by Intel Fortran. Intel Fortran record structures use the RECORD statement and optionally the `STRUCTURE` statement, which are extensions to the Fortran 77 and Fortran standards. The order of data items in a `STRUCTURE` statement determines the order in which the data items are stored.

Intel Fortran stores a record in memory as a linear sequence of values, with the record's first element in the first storage location and its last element in the last storage location. Unless you specify `-align norecords` (Linux) or `/align:norecords` (Windows) padding bytes are added if needed to ensure data fields are naturally aligned.

The following example contains a structure declaration, a RECORD statement, and diagrams of the resulting records as they are stored in memory:

Example

```
STRUCTURE /STRA/
  CHARACTER*1 CHR
  INTEGER*4 INT
END STRUCTURE
...
RECORD /STRA/ REC
```

Figure 1-5 shows the memory diagram of record `REC` for naturally aligned records.

Figure 26: Figure 1-5 Memory Diagram of `REC` for Naturally Aligned Records

Using Arrays Efficiently

This topic discusses how to efficiently access arrays and pass array arguments.

Accessing Arrays Efficiently

Many of the array access efficiency techniques described in this section are applied automatically by the Intel Fortran loop transformation optimizations. Several aspects of array use can improve run-time performance; the following sections discuss the most important aspects.

Perform the fewest operations necessary

The fastest array access occurs when contiguous access to the whole array or most of an array occurs. Perform one or a few array operations that access all of the array or major parts of an array instead of numerous operations on scattered array elements. Rather than use explicit loops for array access, use elemental array operations, such as the following line that increments all elements of array variable `A`:

Example

```
A = A + 1
```

When reading or writing an array, use the array name and not a DO loop or an implied DO-loop that specifies each element number. Fortran 95/90 array syntax allows you to reference a whole array by using its name in an expression.

For example:

Example

```
REAL ::  A(100,100)
A = 0.0
A = A + 1           ! Increment all elements
                    ! of A by 1
...
WRITE (8) A         ! Fast whole array use
```

Similarly, you can use derived-type array structure components, such as:

Example

```
TYPE X
  INTEGER A(5)
END TYPE X
...
TYPE (X) Z
WRITE (8)Z%A        ! Fast array structure
                    ! component use
```

Access arrays using the proper array syntax

Make sure multidimensional arrays are referenced using proper array syntax and are traversed in the natural ascending storage order, which is column-major order for Fortran. With column-major order, the leftmost subscript varies most rapidly with a stride of one. Whole array access uses column-major order.

Avoid row-major order, as is done by C, where the rightmost subscript varies most rapidly. For example, consider the nested `DO` loops that access a two-dimension array with the `J` loop as the innermost loop:

Example

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO I=1,3                  ! I outer loop varies slowest
  DO J=1,5                ! J inner loop varies fastest
    X (I,J) = Y(I,J) + 1
```

Example

```
  ! Inefficient row-major storage order
  END DO                   ! (rightmost subscript varies fastest)
END DO
...
END PROGRAM
```

Since `J` varies the fastest and is the second array subscript in the expression `X (I,J)`, the array is accessed in row-major order. To make the array accessed in natural column-major order, examine the array algorithm and data being modified. Using arrays `X` and `Y`, the array can be accessed in natural column-major order by changing the nesting order of the `DO` loops so the innermost loop variable corresponds to the leftmost array dimension:

Example

```
INTEGER  X(3,5), Y(3,5), I, J
Y = 0
DO J=1,5                   ! J outer loop varies slowest
  DO I=1,3                 ! I inner loop varies fastest
    X (I,J) = Y(I,J) + 1

  ! Efficient column-major storage order
  END DO                   ! (leftmost subscript varies fastest)
END DO
...
END PROGRAM
```

The Intel Fortran whole array access (`X = Y + 1`) uses efficient column major order. However, if the application requires that `J` vary the fastest or if you cannot modify the loop order without changing the results, consider modifying the application to use a rearranged order of array dimensions. Program modifications include rearranging the order of:

- Dimensions in the declaration of the arrays `X(5,3)` and `Y(5,3)`
- The assignment of `X(J,I)` and `Y(J,I)` within the `DO` loops

- All other references to arrays `X` and `Y`

In this case, the original `DO` loop nesting is used where `J` is the innermost loop:

Example

```
INTEGER  X(5,3), Y(5,3), I, J
Y = 0
DO I=1,3                  ! I outer loop varies slowest
  DO J=1,5                ! J inner loop varies fastest
    X (J,I) = Y(J,I) + 1

 ! Efficient column-major storage order
  END DO                  ! (leftmost subscript varies fastest)
END DO
...
END PROGRAM
```

Code written to access multidimensional arrays in row-major order (like C) or random order can often make inefficient use of the CPU memory cache. For more information on using natural storage order during record, see Improving I/O Performance.

Use available intrinsics

Whenever possible, use Fortran array intrinsic procedures instead of creating your own routines to accomplish the same task. Fortran array intrinsic procedures are designed for efficient use with the various Intel Fortran run-time components.

Using the standard-conforming array intrinsics can also make your program more portable.

Avoid leftmost array dimensions

With multidimensional arrays where access to array elements will be noncontiguous, avoid leftmost array dimensions that are a power of two (such as 256, 512).

Since the cache sizes are a power of 2, array dimensions that are also a power of 2 may make inefficient use of cache when array access is noncontiguous. If the cache size is an exact multiple of the leftmost dimension, your program will probably make use of the cache less efficient. This does not apply to contiguous sequential access or whole array access.

One work-around is to increase the dimension to allow some unused elements, making the leftmost dimension larger than actually needed. For example, increasing the leftmost dimension of A from 512 to 520 would make better use of cache:

Example

```
REAL A (512,100)
DO I = 2,511
  DO J = 2,99
    A(I,J)=(A(I+1,J-1) + A(I-1, J+1)) * 0.5
  END DO
END DO
```

In this code, array A has a leftmost dimension of 512, a power of two. The innermost loop accesses the rightmost dimension (row major), causing inefficient access. Increasing the leftmost dimension of `A to 520 (REAL A (520,100))` allows the loop to provide better performance, but at the expense of some unused elements.

Because loop index variables *I* and *J* are used in the calculation, changing the nesting order of the `DO` loops changes the results.

For more information on arrays and their data declaration statements, see the *Intel® Fortran Language Reference*.

Passing Array Arguments Efficiently

In Fortran, there are two general types of array arguments:

- Explicit-shape arrays (introduced with Fortran 77); for example, `A(3,4)` and `B(0:*)`

These arrays have a fixed rank and extent that is known at compile time. Other dummy argument (receiving) arrays that are not deferred-shape (such as assumed-size arrays) can be grouped with explicit-shape array arguments.

- Deferred-shape arrays (introduced with Fortran 95/90); for example, `C(:,:)`

Types of deferred-shape arrays include array pointers and allocatable arrays. Assumed-shape array arguments generally follow the rules about passing deferred-shape array arguments.

When passing arrays as arguments, either the starting (base) address of the array or the address of an array descriptor is passed:

- When using explicit-shape (or assumed-size) arrays to receive an array, the starting address of the array is passed.
- When using deferred-shape or assumed-shape arrays to receive an array, the address of the array descriptor is passed (the compiler creates the array descriptor).

Passing an assumed-shape array or array pointer to an explicit-shape array can slow run-time performance. This is because the compiler needs to create an array temporary for the entire array. The array temporary is created because the passed array may not be contiguous and the receiving (explicit-shape) array requires a contiguous array. When an array temporary is created, the size of the passed array determines whether the impact on slowing run-time performance is slight or severe.

The following table summarizes what happens with the various combinations of array types. The amount of run-time performance inefficiency depends on the size of the array.

	Dummy Argument Array Types (choose one)	
Actual Argument Array Types (choose one)	Explicit-Shape Arrays	Deferred-Shape and Assumed-Shape Arrays
Explicit-Shape Arrays	Result when using this combination: Very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional.	Result when using this combination: Efficient. Only allowed for assumed-shape arrays (not deferred-shape arrays). Does not use an array temporary. Passes an array descriptor. Requires an interface block.

Deferred-Shape and Assumed-Shape Arrays	Result when using this combination: When passing an allocatable array, very efficient. Does not use an array temporary. Does not pass an array descriptor. Interface block optional. When not passing an allocatable array, not efficient. Instead use allocatable arrays whenever possible. Uses an array temporary. Does not pass an array descriptor. Interface block optional.	Result when using this combination: Efficient. Requires an assumed-shape or array pointer as dummy argument. Does not use an array temporary. Passes an array descriptor. Requires an interface block.

Improving I/O Performance

Improving overall I/O performance can minimize both device I/O and actual CPU time. The techniques listed in this topic can significantly improve performance in many applications.

An I/O flow problems limit the maximum speed of execution by being the slowest process in an executing program. In some programs, I/O is the bottleneck that prevents an improvement in run-time performance. The key to relieving I/O problems is to reduce the actual amount of CPU and I/O device time involved in I/O.

The problems can be caused by one or more of the following:

- A dramatic reduction in CPU time without a corresponding improvement in I/O time.
- Such coding practices as:
- - Unnecessary formatting of data and other CPU-intensive processing
  - Unnecessary transfers of intermediate results
  - Inefficient transfers of small amounts of data
  - Application requirements

Improved coding practices can minimize actual device I/O, as well as the actual CPU time. Intel offers software solutions to system-wide problems like minimizing device I/O delays.

Use Unformatted Files Instead of Formatted Files

Use unformatted files whenever possible. Unformatted I/O of numeric data is more efficient and more precise than formatted I/O. Native unformatted data does not need to be modified when transferred and will take up less space on an external file.

Conversely, when writing data to formatted files, formatted data must be converted to character strings for output, less data can transfer in a single operation, and formatted data may lose precision if read back into binary form.

To write the array `A(25,25)` in the following statements, `S1` is more efficient than `S2`:

Example

```
S1          WRITE (7) A
S2          WRITE (7,100) A
100    FORMAT (25(' ',25F5.21))
```

Although formatted data files are more easily ported to other systems, Intel Fortran can convert unformatted data in several formats; see Little-endian-to-Big-endian Conversion (IA-32 architecture).

Write Whole Arrays or Strings

To eliminate unnecessary overhead, write whole arrays or strings at one time rather than individual elements at multiple times. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in implied-`DO` loops. When accessing whole arrays, use the array name (Fortran array syntax) instead of using implied-`DO` loops.

Write Array Data in the Natural Storage Order

Use the natural ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1. (See Accessing Arrays Efficiently.) If a program must read or write data in any other order, efficient block moves are inhibited.

If the whole array is not being written, natural storage order is the best order possible.

If you must use an unnatural storage order, in certain cases it might be more efficient to transfer the data to memory and reorder the data before performing the I/O operation.

Use Memory for Intermediate Results

Performance can improve by storing intermediate results in memory rather than storing them in a file on a peripheral device. One situation that may not benefit from using intermediate storage is when there is a disproportionately large amount of data in relation to physical memory on your system. Excessive page faults can dramatically impede virtual memory performance.

Linux*: If you are primarily concerned with the CPU performance of the system, consider using a memory file system (mfs) virtual disk to hold any files your code reads or writes.

Enable Implied-DO Loop Collapsing

`DO` loop collapsing reduces a major overhead in I/O processing. Normally, each element in an I/O list generates a separate call to the Intel Fortran run-time library (RTL). The processing overhead of these calls can be most significant in implied-`DO` loops.

Intel Fortran reduces the number of calls in implied-`DO` loops by replacing up to seven nested implied-`DO` loops with a single call to an optimized run-time library I/O routine. The routine can transmit many I/O elements at once.

Loop collapsing can occur in formatted and unformatted I/O, but only if certain conditions are met:

- The control variable must be an integer. The control variable cannot be a dummy argument or contained in an `EQUIVALENCE` or `VOLATILE` statement. Intel Fortran must be able to determine that the control variable does not change unexpectedly at run time.
- The format must not contain a variable format expression.

For information on the `VOLATILE` attribute and statement, see the *Intel© Fortran Language Reference*.

Use of Variable Format Expressions

Variable format expressions (an Intel Fortran extension) are almost as flexible as run-time formatting, but they are more efficient because the compiler can eliminate run-time parsing of the I/O format. Only a small amount of processing and the actual data transfer are required during run time.

On the other hand, run-time formatting can impair performance significantly. For example, in the following statements, `S1` is more efficient than `S2` because the formatting is done once at compile time, not at run time:

Example

```
S1        WRITE (6,400) (A(I), I=1,N)
400

 FORMAT (1X, <N> F5.2)
...
S2        WRITE (CHFMT,500) '(1X,',N,'F5.2)'
500

 FORMAT (A,I3,A)
WRITE (6,FMT=CHFMT) (A(I), I=1,N)
```

Efficient Use of Record Buffers and Disk I/O

Records being read or written are transferred between the user's program buffers and one or more disk block I/O buffers, which are established when the file is opened by the Intel Fortran RTL. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimizing physical disk I/O.

You can specify the size of the disk block physical I/O buffer by using the `OPEN` statement `BLOCKSIZE` specifier; the default size can be obtained from `fstat(2)`. If you omit the `BLOCKSIZE` specifier in the `OPEN` statement, it is set for optimal I/O use with the type of device the file resides on (with the exception of network access).

The `OPEN` statement `BUFFERCOUNT` specifier specifies the number of I/O buffers. The default for `BUFFERCOUNT` is 1. Any experiments to improve I/O performance should increase the `BUFFERCOUNT` value and not the `BLOCKSIZE` value, to increase the amount of data read by each disk I/O.

If the `OPEN` statement has `BLOCKSIZE` and `BUFFERCOUNT` specifiers, then the internal buffer size in bytes is the product of these specifiers. If the `open` statement does not have these specifiers, then the default internal buffer size is 8192 bytes. This internal buffer will grow to hold the largest single record, but will never shrink.

The default for the Fortran run-time system is to use unbuffered disk writes. That is, by default, records are written to disk immediately as each record is written instead of accumulating in the buffer to be written to disk later.

To enable buffered writes (that is, to allow the disk device to fill the internal buffer before the buffer is written to disk), use one of the following:

- The `OPEN` statement `BUFFERED` specifier
- The `-assume buffered_io` (Linux*) or `/assume:buffered_io` (Windows*) option of the `ASSUME` command
- The `FORT_BUFFERED` run-time environment variable

The `OPEN` statement `BUFFERED` specifier takes precedence over the `-assume buffered_io` (Linux) or `/assume:buffered_io` (Windows) option. If neither one is set (which is the default), the `FORT_BUFFERED` environment variable is tested at run time.

The `OPEN` statement `BUFFERED` specifier applies to a specific logical unit. In contrast, the `-assume nobuffered_io` (Linux) or `/assume:nobuffered_io` (Windows) option and the `FORT_BUFFERED` environment variable apply to all Fortran units.

Using buffered writes usually makes disk I/O more efficient by writing larger blocks of data to the disk less often. However, a system failure when using buffered writes can cause records to be lost, since they might not yet have been written to disk. (Such records would have been written to disk with the default unbuffered writes.)

When performing I/O across a network, be aware that the size of the block of network data sent across the network can impact application efficiency. When reading network data, follow the same advice for efficient disk reads, by increasing the `BUFFERCOUNT`. When writing data through the network, several items should be considered:

- Unless the application requires that records be written using unbuffered writes, enable buffered writes by a method described above.
- Especially with large files, increasing the `BLOCKSIZE` value increases the size of the block sent on the network and how often network data blocks get sent.
- Time the application when using different `BLOCKSIZE` values under similar conditions to find the optimal network block size.

Completion of a `WRITE` statement, even when not buffered, does not guarantee that the data has been written to disk; the operating system may delay the actual disk write. To ensure that the data has been written to disk, you can call the `FLUSH` routine. Because this can cause a significant slowdown of the application, `FLUSH` should be used only when absolutely needed.

Linux* Only: When writing records, be aware that I/O records are written to unified buffer cache (UBC) system buffers. To request that I/O records be written from program buffers to the UBC system buffers, use the FLUSH library routine. Be aware that calling FLUSH also discards read-ahead data in user buffer. For more information, see `FLUSH` in the *Intel® Fortran Libraries Reference.*

Specify RECL

The sum of the record length (`RECL` specifier in an `OPEN` statement) and its overhead is a multiple or divisor of the blocksize, which is device-specific. For example, if the `BLOCKSIZE` is 8192 then `RECL` might be 24576 (a multiple of 3) or 1024 (a divisor of 8).

The `RECL` value should fill blocks as close to capacity as possible (but not over capacity). Such values allow efficient moves, with each operation moving as much data as possible; the least amount of space in the block is wasted. Avoid using values larger than the block capacity, because they create very inefficient moves for the excess data only slightly filling a block (allocating extra memory for the buffer and writing partial blocks are inefficient).

The `RECL` value unit for formatted files is always 1-byte units. For unformatted files, the `RECL` unit is 4-byte units, unless you specify the `-assume byterecl` (Linux) or `/assume:byterecl` (Windows) option for the `ASSUME` specifier to request 1-byte units.

Use the Optimal Record Type

Unless a certain record type is needed for portability reasons, choose the most efficient type, as follows:

- For sequential files of a consistent record size, the fixed-length record type gives the best performance.
- For sequential unformatted files when records are not fixed in size, the variable-length record type gives the best performance, particularly for `BACKSPACE` operations.
- For sequential formatted files when records are not fixed in size, the `Stream_LF` record type gives the best performance.

Reading from a Redirected Standard Input File

Due to certain precautions that the Fortran run-time system takes to ensure the integrity of standard input, reads can be very slow when standard input is redirected from a file. For example, when you use a command such as `myprogram.exe < myinput.data>` , the data is read using the `READ(*)` or `READ(5)` statement, and performance is degraded. To avoid this problem, do one of the following:

- Explicitly open the file using the `OPEN` statement. For example: `OPEN(5, STATUS='OLD', FILE='myinput.dat')`
- Use an environment variable to specify the input file.

To take advantage of these methods, be sure your program does not rely on sharing the standard input file.

For more information on Intel Fortran data files and I/O, see Files, Devices, and I/O in the Building Applications; on `OPEN` statement specifiers and defaults, see Open Statement in the *Intel® Fortran Language Reference*.

Improving Run-time Efficiency

Use these source-coding guidelines to improve run-time performance. The amount of improvement in run-time performance is related to the number of times a statement is executed. For example, improving an arithmetic expression executed within a loop many times has the potential to improve performance, more than improving a similar expression executed once outside a loop.

Avoid Small Integer and Small Logical Data Items

Avoid using integer or logical data less than 32 bits. Accessing a 16-bit (or 8-bit) data type can make data access less efficient, especially on IA-64 architecture based systems.

To minimize data storage and memory cache misses with arrays, use 32-bit data rather than 64-bit data, unless you require the greater numeric range of 8-byte integers or the greater range and precision of double precision floating-point numbers.

Avoid Using Slow Arithmetic Operators

Before you modify source code to avoid slow arithmetic operators, be aware that optimizations convert many slow arithmetic operators to faster arithmetic operators. For example, the compiler optimizes the expression `H=J**2` to be `H=J*J`.

Consider also whether replacing a slow arithmetic operator with a faster arithmetic operator will change the accuracy of the results or impact the maintainability (readability) of the source code.

Replacing slow arithmetic operators with faster ones should be reserved for critical code areas. The following list shows the Intel Fortran arithmetic operators, from fastest to slowest:

1. Addition (+), Subtraction (-), and Floating-point multiplication (*)
2. Integer multiplication (*)
3. Division (/)
4. Exponentiation (**)

Avoid Using EQUIVALENCE Statements

Avoid using `EQUIVALENCE` statements. `EQUIVALENCE` statements can:

- Force unaligned data or cause data to span natural boundaries.
- Prevent certain optimizations, including:
  - Global data analysis under certain conditions; see the `-O2` (Linux*) or `/O2` (Windows*) option description for more information.
  - Implied-`DO` loop collapsing when the control variable is contained in an `EQUIVALENCE` statement

Use Statement Functions and Internal Subprograms

Whenever the Intel compiler has access to the use and definition of a subprogram during compilation, it may choose to inline the subprogram. Using statement functions and internal subprograms maximizes the number of subprogram references that will be inlined, especially when multiple source files are compiled together at optimization level `-O3` (Linux) or `/O3` (Windows).

For more information, see Efficient Compilation.

Code DO Loops for Efficiency

Minimize the arithmetic operations and other operations in a `DO` loop whenever possible. Moving unnecessary operations outside the loop will improve performance (for example, when the intermediate nonvarying values within the loop are not needed).

Using Fortran Intrinsics

Intel® Fortran supports all standard Fortran intrinsic procedures and provides Intel-specific intrinsic procedures to extend the language functionality. The Intel-specific intrinsic procedures are provided in `libifcore.lib`. For more details on these intrinsic procedures, see the *Intel® Fortran Language Reference*.

This topic provides an example of an Intel-extended intrinsic that is helpful in developing efficient applications on IA-32, Intel® 64, and IA-64 architectures.

CACHESIZE Intrinsic

The intrinsic `CACHESIZE(n)` returns the size in kilobytes of the cache at level `n`; one (1) represents the first level cache. Zero (0) is returned for a nonexistent cache level.

Use this intrinsic in any situation you would like to tailor algorithms for the cache hierarchy on the target processor. For example, an application may query the cache size and use the result to select block sizes in algorithms that operate on matrices.

Example

```
subroutine foo (level)
integer level
if (cachesize(level) > threshold) then
   call big_bar()
else
   call small_bar()
end if
end subroutine
```

Understanding Run-time Performance

The information in this topic assumes that you are using a performance optimization methodology and have analyzed the application type you are optimizing.

After profiling your application to determine where best to spend your time, attempt to discover what optimizations and what limitations have been imposed by the compiler. Use the compiler reports to determine what to try next.

Depending on what you discover from the reports you may be able to help the compiler through options, directives, and slight code modifications to take advantage of key architectural features to achieve the best performance.

The compiler reports can describe what actions have been taken and what actions cannot be taken based on the assumptions made by the compiler. Experimenting with options and directives allows you to use an understanding of the assumptions and suggest a new optimization strategy or technique.

Helping the Compiler

You can help the compiler in some important ways:

- Read the appropriate reports to gain an understanding of what the compiler is doing for you and the assumptions the compiler has made with respect to your code.
- Use specific options, intrinsics, libraries, and directives to get the best performance from your application.

Use the Math Kernel Library (MKL) instead of user code, or calling F90 intrinsics instead of user code.

See Applying Optimization Strategies for other suggestions.

Memory Aliasing For IA-64 Architectures

Memory aliasing is the single largest issue affecting the optimizations in the Intel® compiler for IA-64 architecture based systems. Memory aliasing is writing to a given memory location with more than one pointer. The compiler is cautious to not optimize too aggressively in these cases; if the compiler optimizes too aggressively, unpredictable behavior can result (for example, incorrect results, abnormal termination, etc.).

Since the compiler usually optimizes on a module-by-module, function-by-function basis, the compiler does not have an overall perspective with respect to variable use for global variables or variables that are passed into a function; therefore, the compiler usually assumes that any pointers passed into a function are likely to be aliased. The compiler makes this assumption even for pointers you know are not aliased. This behavior means that perfectly safe loops do not get pipelined or vectorized, and performance suffers.

There are several ways to instruct the compiler that pointers are not aliased:

- Use a comprehensive compiler option, such as `-fno-alias` (Linux*) or `/Oa` (Windows*). These options instruct the compiler that no pointers in any module are aliased, placing the responsibility of program correctness directly with the developer.

- Use a less comprehensive option, like `-fno-fnalias` (Linux) or `/Ow` (Windows). These options instruct the compiler that no pointers passed through function arguments are aliased.

  Function arguments are a common example of potential aliasing that you can clarify for the compiler. You may know that the arguments passed to a function do not alias, but the compiler is forced to assume so. Using these options tells the compiler it is now safe to assume that these function arguments are not aliased. This option is still a somewhat bold statement to make, as it affects all functions in the module(s) compiled with the `-fno-nalias` (Linux) or `/Ow` (Windows) option.
- Use the `IDVEP` directive. Alternatively, you might use a directive that applies to a specified loop in a function. This is more precise than specifying an entire function. The directive asserts that, for a given loop, there are no vector dependencies. Essentially, this is the same as saying that no pointers are aliasing in a given loop.

Non-Unit Stride Memory Access

Another issue that can have considerable impact on performance is accessing memory in a non-Unit Stride fashion. This means that as your inner loop increments consecutively, you access memory from non adjacent locations. For example, consider the following matrix multiplication code:

Example

```
!Non-Unit Stride Memory Access
subroutine non_unit_stride_memory_access(a,b,c, NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
  do i=1,NUM
    do j=1,NUM
      do k=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine non_unit_stride_memory_access
```

Notice that `c[i][j]`, and `a[i][k]` both access consecutive memory locations when the inner-most loops associated with the array are incremented. The *b* array however, with its loops with indexes k and j, does not access Memory Unit Stride. When the loop reads `b[k=0][j=0]` and then the k loop increments by one to `b[k=1][j=0]`, the loop has skipped over NUM memory locations having skipped `b[k][1], b[k][2] .. b[k][NUM].`

Loop transformation (sometimes called loop interchange) helps to address this problem. While the compiler is capable of doing loop interchange automatically, it does not always recognize the opportunity.

The memory access pattern for the example code listed above is illustrated in the following figure:

Assume you modify the example code listed above by making the following changes to introduce loop interchange:

Example

```
subroutine unit_stride_memory_access(a,b,c, NUM)
  implicit none
  integer :: i,j,k,NUM
  real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop after interchange
  do i=1,NUM
    do k=1,NUM
      do j=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
      end do
    end do
  end do
end subroutine unit_stride_memory_access
```

After the loop interchange the memory access pattern might look the following figure:

Understanding Data Alignment

Aligning data on boundaries can help performance. The Intel® compiler attempts to align data on boundaries for you. However, as in all areas of optimization, coding practices can either help or hinder the compiler and can lead to performance problems.

Always attempt to optimize using compiler options first.

To avoid performance problems you should keep the following guidelines in mind, which are separated by architecture:

IA-32, Intel® 64, and IA-64 architectures:

- Do not access or create data at large intervals that are separated by exactly 2^n (for example, 1 KB, 2 KB, 4 KB, 16 KB, 32 KB, 64 KB, 128 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, etc.).
- Align data so that memory accesses does not cross cache lines (for example, 32 bytes, 64 bytes, 128 bytes).
- Use Application Binary Interface (ABI) for the Itanium® compiler to insure that ITP pointers are 16-byte aligned.

IA-32 and Intel® 64 architectures:

- Align data to correspond to the SIMD or Streaming SIMD Extension registers sizes.

IA-64 architecture:

- Avoid using packed structures.
- Avoid casting pointers of small data elements to pointers of large data elements.
- Do computations on unpacked data, then repack data if necessary, to correctly output the data.

In general, keeping data in cache has a better performance impact than keeping the data aligned. Try to use techniques that conform to the rules listed above.

See Setting Data Type and Alignment for more detailed information on aligning data.

Timing Your Application

You can start collecting information about your application performance by timing your application. More sophisticated and helpful data can be collected by using performance analysis tools.

Considerations on Timing Your Application

One of the performance indicators is your application timing. The following considerations apply to timing your application:

- Run program timings when other users are not active. Your timing results can be affected by one or more CPU-intensive processes also running while doing your timings.
- Try to run the program under the same conditions each time to provide the most accurate results, especially when comparing execution times of a previous version of the same program. Use the same system (processor model, amount of memory, version of the operating system, and so on) if possible.

- If you do need to change systems, you should measure the time using the same version of the program on both systems, so you know each system's effect on your timings.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Certain overhead functions like loading libraries might influence short timings considerably.
- If your program displays a lot of text, consider redirecting the output from the program. Redirecting output from the program will change the times reported because of reduced screen I/O.

  Timings that show a large amount of system time may indicate a lot of time spent doing I/O, which might be worth investigating.
- For programs that run for less than a few seconds, run several timings to ensure that the results are not misleading. Overhead functions like loading shared libraries might influence short timings considerably.

  Use the `time` command and specify the name of the executable program to provide the following:

  - The elapsed, real, or "wall clock" time, which will be greater than the total charged actual CPU time.
  - Charged actual CPU time, shown for both system and user execution. The total actual CPU time is the sum of the actual user CPU time and actual system CPU time.

Methods of Timing Your Application

To perform application timings, use a version of the `TIME` command in a `.BAT` file (or the function timing profiling option). You might consider modifying the program to call routines within the program to measure execution time (possibly using conditionally compiled lines).

For example:

- Intel Fortran intrinsic procedures, such as `SECNDS`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`, and `DATE_AND_TIME`.
- Portability library routines, such as `DCLOCK`, `ETIME`, `SECNDS`, or `TIME`.

Whenever possible, perform detailed performance analysis on a system that closely resembles the system(s) that will be used for actual application use.

Sample Timing

The following program template could be run by a `.BAT` file that executes the `TIME` command both before and after execution, to provide an approximate wall-clock time for the execution of the entire program. The Fortran intrinsic `CPU_TIME` can be used at selected points in the program to collect the CPU time between the start and end of the task to be timed.

Example

```
REAL time_begin, time_end
  ...
  CALL CPU_TIME ( time_begin )
  !
  !task to be timed
  !
  CALL CPU_TIME ( time_end )
  PRINT *, 'Time of operation was ', &
time_end - time_begin, ' seconds'
```

Considerations for Linux*

In the following example timings, the sample program being timed displays the following line:

Bourne* shell example

```
Average of all the numbers is:    4368488960.000000
```

Using the Bourne* shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

Bourne* shell example

```
$ time a.out
Average of all the numbers is:
4368488960.000000
real    0m2.46s
user    0m0.61s
sys     0m0.58s
```

Using the C shell, the following program timing reports 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use), about 4 seconds (0:04) of elapsed time, the use of 28% of available CPU time, and other information:

C shell l example

```
% time a.out
Average of all the numbers is:   4368488960.000000
0.61u 0.58s 0:04 28% 78+424k 9+5io 0pf+0w
```

Using the bash shell, the following program timing reports that the program uses 1.19 seconds of total actual CPU time (0.61 seconds in actual CPU time for user program use and 0.58 seconds of actual CPU time for system use) and 2.46 seconds of elapsed time:

bash shell l example

```
[user@system user]$ time ./a.out
Average of all the numbers is:   4368488960.000000
elapsed  0m2.46s
user     0m0.61s
sys      0m0.58s
```

Timings that indicate a large amount of system time is being used may suggest excessive I/O, a condition worth investigating.

If your program displays a lot of text, you can redirect the output from the program on the time command line. Redirecting output from the program will change the times reported because of reduced screen I/O.

For more information, see `time(1)`.

In addition to the `time` command, you might consider modifying the program to call routines within the program to measure execution time. For example, use the Intel intrinsic procedures, such as `SECNDS`, `DCLOCK`, `CPU_TIME`, `SYSTEM_CLOCK`, `TIME`, and `DATE_AND_TIME`.

Applying Optimization Strategies

The compiler may or may not apply the following optimizations to your loop: Interchange, Unrolling, Cache Blocking, and LoadPair. These transformations are discussed in the following sections, including how to transform loops manually and how to control them with directives or internal options.

Loop Interchange

Loop Interchange is a nested loop transformation applied by High-level Optimization (HLO) that swaps the order of execution of two nested loops. Typically, the transformation is performed to provide sequential Unit Stride access to array elements used inside the loop to improve cache locality. The compiler `-O3` (Linux* and Mac OS* X) or `/O3` (Windows*) optimization looks for opportunities to apply loop interchange for you.

The following is an example of a loop interchange

Example

```
subroutine loop_interchange(a,b,c, NUM)
implicit none
integer :: i,j,k,NUM
real :: a(NUM,NUM), b(NUM,NUM), c(NUM,NUM)
! loop before loop interchange
do i=1,NUM
  do j=1,NUM
    do k=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
    end do
  end do
end do
! loop after interchange
do i=1,NUM
  do k=1,NUM
    do j=1,NUM
        c(j,i) = c(j,i) + a(j,k) * b(k,i)
    end do
  end do
end do
end subroutine loop_interchange
```

Unrolling

Loop unrolling is a loop transformation generally used by HLO that can take better advantage of Instruction-Level Parallelism (ILP), keeping as many functional units busy doing useful work as possible during single loop iteration. In loop unrolling, you add more work to the inside of the loop while doing fewer loop iterations in exchange.

Example

```
subroutine loop_unroll_before(a,b,c,N,M)
implicit none
integer :: i,j,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
do i=1,N
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
  end do
end do
end subroutine loop_unroll_before
```

Example

```
subroutine loop_unroll_after(a,b,c,N,M)
implicit none
integer :: i,j,K,N,M
real :: a(N,M), b(N,M), c(N,M)
N=1025
M=5
K=MOD(N,4)  !K= N MOD 4
! main part of loop
```

Example

```
do i=1,N-K,4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
    a(j,i+1) = b(j,i+1) + c(j,i+1)
    a(j,i+2) = b(j,i+2) + c(j,i+2)
    a(j,i+3) = b(j,i+3) + c(j,i+3)
  end do
end do
! post conditioning part of loop
do i= N-K+2, N, 4
  do j=1,M
    a(j,i) = b(j,i) + c(j,i)
  end do
end do
end subroutine loop_unroll_after
```

Post conditioning is preferred over pre-conditioning because post conditioning will preserve the data alignment and avoid the cost of memory alignment access penalties.

Cache Blocking

Cache blocking involves structuring data blocks so that they conveniently fit into a portion of the L1 or L2 cache. By controlling data cache locality, an application can minimize performance delays due to memory bus access. The application controls the behavior by dividing a large array into smaller blocks of memory so a thread can make repeated accesses to the data while the data is still in cache.

For example, image processing and video applications are well suited to cache blocking techniques because an image can be processed on smaller portions of the total image or video frame. Compilers often use the same technique, by grouping related blocks of instructions close together so they execute from the L2 cache.

The effectiveness of the cache blocking technique depends on data block size, processor cache size, and the number of times the data is reused. Cache sizes vary based on processor. An application can detect the data cache size using the `CPUID` instruction and dynamically adjust cache blocking tile sizes to maximize performance. As a general rule, cache block sizes should target approximately one-half to three-quarters the size of the physical cache. For systems that are Hyper-Threading Technology (HT Technology) enabled target one-quarter to one-half the physical cache size. Designing for Hyper-Threading Technology

Cache blocking is applied in HLO and is used on large arrays where the arrays cannot all fit into cache simultaneously. This method is one way of pulling a subset of data into cache (in a small region), and using this cached data as effectively as possible before the data is replaced by new data from memory.

Example

```
subroutine cache_blocking_before(a,b,N)
  implicit none
  integer :: i,j,k,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do i = 1, N
    do j = 1, N
      do k = 1, N
        a(i,j,k) = a(i,j,k)  + b(i,j,k)
      end do
    end do
  end do
end subroutine cache_blocking_before
subroutine cache_blocking_after(a,b,N)
  implicit none
  integer :: i,j,k,u,v,N
  real :: a(N,N,N), b(N,N,N), c(N,N,N)
  N=1000
  do v = 1, N, 20
    do u = 1, N, 20
      do k = v, v+19
        do j = u, u+19
          do i = 1, N
            a(i,j,k) = a(i,j,k)  + b(i,j,k)
```

Example

```
          end do
        end do
      end do
    end do
  end do
end subroutine cache_blocking_after
```

The cache block size is set to 20. The goal is to read in a block of cache, do every bit of computing we can with the data in this cache, then load a new block of data into cache. There are 20 elements of `A` and 20 elements of `B` in cache at the same time and you should do as much work with this data as you can before you increment to the next cache block.

Blocking factors will be different for different architectures. Determine the blocking factors experimentally. For example, different blocking factors would be required for single precision versus double precision. Typically, the overall impact to performance can be significant.

Loop Distribution

Loop distribution is a high-level loop transformation that splits a large loop into two smaller loops. It can be useful in cases where optimizations like software-pipelining (SWP) or vectorization cannot take place due to excessive register usage. By splitting a loop into smaller segments, it may be possible to get each smaller loop or at least one of the smaller loops to SWP or vectorize. An example is as follows:

Example

```
subroutine loop_distribution_before(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_before
subroutine loop_distribution_after(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
```

Example

```
    c(i) = c(i)  + i
  end do
  do i = 1, N
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_after
```

There are directives to suggest distributing loops to the compiler as follows:

Example

```
!DEC$ distribute point
```

Placed outside a loop, the compiler will attempt to distribute the loop based on an internal heuristic. The following is an example of using the pragma outside the loop:

Example

```
subroutine loop_distribution_pragma1(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
!DEC$ distribute point
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_pragma1
```

Placed within a loop, the compiler will attempt to distribute the loop at that point. All loop-carried dependencies will be ignored. The following example uses the directive within a loop to precisely indicate where the split should take place:

Example

```
subroutine loop_distribution_pragma2(a,b,c,x,y,z,N)
  implicit none
  integer :: i,N
  real :: a(N), b(N), c(N), x(N), y(N), z(N)
  N=1024
  do i = 1, N
    a(i) = a(i)  + i
    b(i) = b(i)  + i
    c(i) = c(i)  + i
!DEC$  distribute point
    x(i) = x(i)  + i
    y(i) = y(i)  + i
    z(i) = z(i)  + i
  end do
end subroutine loop_distribution_pragma2
```

Load Pair (Itanium® Compiler)

Load pairs (ldfp) are instructions that load two contiguous single or double precision values from memory in one move. Load pairs can significantly improve performance.

Manual Loop Transformations

There might be cases where these manual transformations are called acceptable or even preferred. As a general rule, you should let the compiler transform loops for you. Manually transform loops as a last resort; use this strategy only in cases where you are attempting to gain performance increases.

Manual loop transformations have many disadvantages, which include the following:

- Application code becomes harder to maintain over time.
- New compiler features can cause you to lose any optimization you gain by manually transforming the loop.
- Architectural requirements might restrict your code to a specific architecture unintentionally.

The `HLO` report can give you an idea of what loop transformations have been applied by the compiler.

Experimentation is a critical component in manually transforming loops. You might try to apply a loop transformation that the compiler ignored. Sometimes, it is beneficial to apply a manual loop transformation that the compiler has already applied with `-O3` (Linux) or `/O3` (Windows).

Optimizing the Compilation Process

Optimizing the Compilation Process Overview

This section describes the Intel® compiler options that can optimize the compilation process. By default, the compiler converts source code directly to an executable file. Appropriate options enable you not only to control the process and obtain desired output file produced by the compiler, but also make the compilation itself more efficient.

A group of options monitors the outcome of Intel compiler-generated code without interfering with the way your program runs. These options control some computation aspects, such as allocating the stack memory, setting or modifying variable settings, and defining the use of some registers.

The options in this section provide you with the following capabilities of efficient compilation:

- Compiling efficiently
- Using default compiler optimizations
- Automatic allocation of variables and stacks
- Converting little-endian to big-endian
- Symbol visibility attribute options

Efficient Compilation

Efficient compilation contributes to performance improvement. Before you analyze your program for performance improvement, and improve program performance, you should think of efficient compilation itself.

Based on the analysis of your application, you can decide which compiler optimizations and command-line options can improve the run-time performance of your application.

Efficient Compilation Techniques

The efficient compilation techniques can be used during the earlier stages and later stages of program development. During the earlier stages of program development, you can use incremental compilation without optimization. For example:

Linux* and Mac OS* X

Example

`ifort -c -g -O0 sub2.f90`
`ifort -c -g -O0 sub3.f90`
`ifort -o main -g -O0 main.f90 sub2.o sub3.o`

The above commands turn off all compiler default optimizations, for example, `-O2` (Linux* and Mac OS* X) or `/O2` (Windows*), with `-O0` (Linux and Mac OS X) or `/Od` (Windows). You can use the `-g` (Linux) or `/Zi` or `/debug:full` (Windows) option to generate symbolic debugging information and line numbers in the object code for all routines in the program for use by a source-level debugger. The `main` file created in the third command above contains symbolic debugging information as well.

During the later stages of program development, you should specify multiple source files together and use an optimization level of at least `-O2` (Linux and Mac OS X) or `/O2` (Windows) to allow more optimizations to occur. For instance, the following command compiles all three source files together using the default level of optimization:

Linux and Mac OS X

Example
`ifort -o main main.f90 sub2.f90 sub3.f90`

Compiling multiple source files lets the compiler examine more code for possible optimizations, which results in:

- Inlining more procedures
- More complete data flow analysis
- Reducing the number of external references to be resolved during linking

For very large programs, compiling all source files together may not be practical. In such instances, consider compiling source files containing related routines together using multiple `ifort` commands, rather than compiling source files individually.

Stacks: Automatic Allocation and Checking

The options in this group enable you to control the computation of stacks and variables in the compiler generated code.

Automatic Allocation of Variables

`-automatic` (Linux*and Mac OS* X) and `/automatic` (Windows*)

These options specify that locally declared variables are allocated to the run-time stack rather than static storage. If variables defined in a procedure do not have the `SAVE` or `ALLOCATABLE` attribute, they are allocated to the stack. It does not affect variables that appear in an `EQUIVALENCE` or `SAVE` statement, or those that are in `COMMON`.

`-automatic` (Linux and Mac OS X) or `/automatic` (Windows) may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across routine calls should appear in a `SAVE` statement.

NOTE. Linux: If you specify `-recursive` or `-openmp`, the default is `-auto`.

Windows: The Windows NT* system imposes a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/Qauto` because arrays are allocated on the stack along with scalars.

`-auto-scalar` (Linux and Mac OS X) or `/Qauto-scalar` (Windows)

These options cause allocation of local scalar variables of intrinsic type `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL` to the stack. This option does not affect variables that appear in an `EQUIVALENCE` or `SAVE` statement, or those that are in `COMMON`.

The `-auto-scalar` (Linux and Mac OS X) or `/Qauto-scalar` (Windows) option may provide a performance gain for your program, but if your program depends on variables having the same value as the last time the routine was invoked, your program may not function properly. Variables that need to retain their values across subroutine calls should appear in a `SAVE` statement. This option is similar to `-auto` (Linux and Mac OS X) and `/Qauto` (Windows) which

causes all local variables to be allocated on the stack. The difference is that `-auto-scalar` (Linux and Mac OS X) or `/Qauto-scalar` (Windows) allocates only scalar variables of the stated above intrinsic types to the stack.

NOTE. Windows: Windows NT* imposes a performance penalty for addressing a stack frame that is too large. This penalty may be incurred with `/Qauto` because arrays are allocated on the stack along with scalars. However, with `/Qauto-scalar`, you would have to have more than 32K bytes of local scalar variables before you incurred the performance penalty.

`-auto-scalar` (Linux and Mac OS X) or `/Qauto-scalar` (Windows) enables the compiler to make better choices about which variables should be kept in registers during program execution.

`-save`, `-zero[-]` (Linux and Mac OS X) or `/Qsave`, `/Qzero[-]` (Windows)

The `-save` (Linux and Mac OS X) or `/Qsave` (Windows) option is opposite of `-auto` (Linux) or `/Qauto` (Windows). The `-save` (Linux and Mac OS X) or `/Qsave` (Windows) option saves all variables in static allocation except local variables within a recursive routine.

If a routine is invoked more than once, this option forces the local variables to retain their values from the last invocation. The `save` option ensures that the final results on the exit of the routine is saved on memory and can be reused at the next occurrence of that routine. This may cause some performance degradation as it causes more frequent rounding of the results.

NOTE. Linux and Mac OS X: `-save` is the same as `-noauto`.

Windows: `/Qsave` is the same as `/save`, and `/noautomatic`.

When the compiler optimizes the code, the results are stored in registers.

The `-zero[-]` (Linux and Mac OS X) or `/Qzero[-]` (Windows) option initializes to zero all local scalar variables of intrinsic type `INTEGER`, `REAL`, `COMPLEX`, or `LOGICAL`, which are saved and not initialized yet. Used in conjunction with `SAVE`.

Summary

There are three options for allocating variables: `-save` (Linux and Mac OS X) or `/Qsave` (Windows), `-auto` (Linux) or `/Qauto` (Windows) and `-auto-scalar` (Linux) or `/Qauto-scalar` (Windows). Only one of these three can be specified.

The correlation among them is as follows:

- -save (Linux and Mac OS X) or /Qsave (Windows) disables -auto (Linux and Mac OS X) or /Qauto (Windows), sets -noauto (Linux and Mac OS X) or /noautomatic (Windows), and allocates all variables not marked AUTOMATIC to static memory.
- -auto (Linux and Mac OS X) or /Qauto (Windows) disables -save (Linux and Mac OS X) or /Qsave (Windows), sets -nosave (Linux and Mac OS X) or /automatic (Windows) and allocates all variables, scalars and arrays of all types, not marked SAVE to the stack.
- -auto-scalar (Linux and Mac OS X) or /Qauto-scalar (Windows) makes local scalars of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL automatic. Additionally, this is the default. There is no -noauto-scalar (Linux and Mac OS X) or /Qauto-scalar- (Windows); however, -recursive or -openmp (Linux and Mac OS X) or /recursive or /Openmp (Windows) disables -auto-scalar (Linux and Mac OS X) or /Qauto-scalar (Windows) and makes -auto (Linux and Mac OS X) or /Qauto (Windows) the default.

Checking and Setting Space

The following options perform checking and setting space for stacks (these options are supported on Windows only):

- The /Gs0 option enables stack-checking for all functions.
- The /Gs*n* option checks by default the stack space allocated for functions with more than 4KB.
- The /F*n* option sets the stack reserve amount for the program. The /F*n* option passes /stack:*n* to the linker.

Aliases

The -common-args (Linux and Mac OS X) or /Qcommon-args (Windows) option assumes that the by-reference subprogram arguments may have aliases of one another.

It is recommended that you use the -assume dummy_aliases (Linux and Mac OS X) or /assume:dummy_aliases (Windows) option instead of the common-args option.

For more information about using the preferred option, see the following topic:

- -assume compiler option

Preventing CRAY* Pointer Aliasing

Option `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) specifies that the CRAY* pointers do not alias with other variables. Consider the following example.

Example

```
pointer (pb, b)
pb = getstorage()
do i = 1, n
  b(i) = a(i) + 1
enddo
```

When the option is not specified, the compiler assumes that b and a are aliased. To prevent such an assumption, specify this option, and the compiler will treat `b(i)` and `a(i)` as independent of each other.

However, if the variables are intended to be aliased with CRAY pointers, using the `-safe-cray-ptr` (Linux and Mac OS X) or `/Qsafe-cray-ptr` (Windows) option produces incorrect result. For the code example below, the option should not be used.

Example

```
pb = loc(a(2))
do i=1, n
  b(i) = a(i) +1
enddo
```

Cross Platform

For example, an object of type real cannot be accessed as an integer. You should see the ANSI standard for the complete set of rules.

The option directs the compiler to assume the following:

- Arrays are not accessed out of arrays' bounds.
- Pointers are not cast to non-pointer types and vice-versa.

- References to objects of two different scalar types cannot alias. For example, an object of type integer cannot alias with an object of type real or an object of type real cannot alias with an object of type double precision.

If your program satisfies the above conditions, setting this option will help the compiler better optimize the program. However, if your program may not satisfy one of the above conditions, the option must be disabled, as it can lead the compiler to generate incorrect code.

For more information, see the following topic:

- `-ansi-alias` compiler option

Little-endian-to-Big-endian Conversion (IA-32 Architecture)

The Intel® compiler can write unformatted sequential files in big-endian format and can also read files produced in big-endian format by using the little-endian-to-big-endian conversion feature.

On processors based on IA-32 or IA-64 architectures, Intel Fortran handles internal data in little-endian format. The little-endian-to-big-endian conversion feature is intended for Fortran unformatted input/output operations in unformatted sequential files. The feature enables the following:

- Processing of the files developed on processors that accept big-endian data format
- Producing big-endian files for such processors on little-endian systems.

The little-endian-to-big-endian conversion is accomplished by the following operations:

- The `WRITE` operation converts little-endian format to big-endian format.
- The `READ` operation converts big-endian format to little-endian format.

The feature enables the conversion of variables and arrays (or array subscripts) of basic data types. Derived data types are not supported.

Little-to-Big Endian Conversion Environment Variable

In order to use the little-endian-to-big-endian conversion feature, specify the numbers of the units to be used for conversion purposes by setting the `F_UFMTENDIAN` environment variable. Then, the `READ`/`WRITE` statements that use these unit numbers, will perform relevant conversions. Other `READ`/`WRITE` statements will work in the usual way.

In the general case, the variable consists of two parts divided by a semicolon. No spaces are allowed inside the `F_UFMTENDIAN` value. The variable has the following syntax:

Example
`F_UFMTENDIAN=MODE	[MODE;] EXCEPTION`

where the following conditions are true:

Conditions
`MODE = big	little`
`EXCEPTION = big:ULIST	little:ULIST	ULIST`
`ULIST = U	ULIST,U`
`U = decimal	decimal -decimal`

and the following conditions apply:

- `MODE` defines current format of data, represented in the files; it can be omitted.

  The keyword `little` means that the data have little endian format and will not be converted. This keyword is a default.

  The keyword `big` means that the data have big endian format and will be converted. This keyword may be omitted together with the colon.
- `EXCEPTION` is intended to define the list of exclusions for `MODE`; it can be omitted. `EXCEPTION` keyword (`little` or `big`) defines data format in the files that are connected to the units from the `EXCEPTION` list. This value overrides `MODE` value for the units listed.
- Each list member `U` is a simple unit number or a number of units. The number of list members is limited to 64.

  `decimal` is a non-negative decimal number less than 2^{32}.

Converted data should have basic data types, or arrays of basic data types. Derived data types are disabled.

Command lines for variable setting with different shells:

Shell	Command Line
Sh	`export F_UFMTENDIAN=MODE;EXCEPTION`
Csh	`setenv F_UFMTENDIAN MODE;EXCEPTION`

NOTE. Environment variable value should be enclosed in quotes if semicolon is present.

Another Possible Environment Variable Setting

The environment variable can also have the following syntax:

Example
`F_UFMTENDIAN=u[,u] . . .`

Command lines for the variable setting with different shells:

Shell	Command Line
Sh	`export F_UFMTENDIAN=u[,u] . . .`
Csh	`setenv F_UFMTENDIAN u[,u] . . .`

See error messages that may be issued during the little-endian to big-endian conversion. They are all fatal. You should contact Intel if such errors occur.

Usage Examples

The following usage examples illustrate the concepts detailed above.

Example 1
`F_UFMTENDIAN=big`

All input/output operations perform conversion from big-endian to little-endian on `READ` and from little-endian to big-endian on `WRITE`.

Example 2

`F_UFMTENDIAN="little;big:10,20"`
or
`F_UFMTENDIAN=big:10,20`
or

Example 2

```
F_UFMTENDIAN=10,20
```

In this case, only on unit numbers 10 and 20 the input/output operations perform big-little endian conversion.

Example 3

```
F_UFMTENDIAN="big;little:8"
```

In this case, on unit number 8 no conversion operation occurs. On all other units, the input/output operations perform big-little endian conversion.

Example 4

```
4. F_UFMTENDIAN=10-20
```

Define 10, 11, 12...19, 20 units for conversion purposes; on these units, the input/output operations perform big-little endian conversion.

Assume you set `F_UFMTENDIAN`=10,100 and run the following program.

Example 5

```
integer*4   cc4
integer*8   cc8
integer*4   c4
integer*8   c8
c4 = 456
c8 = 789
```

Example 5

```
C  prepare a little endian representation of data
open(11,file='lit.tmp',form='unformatted')
write(11) c8
write(11) c4
close(11)
C  prepare a big endian representation of data
open(10,file='big.tmp',form='unformatted')
write(10) c8
write(10) c4
close(10)
C  read big endian data and operate with them on
C  little endian machine.
open(100,file='big.tmp',form='unformatted')
read(100) cc8
read(100) cc4
C    Any operation with data, which have been read
C
. . .
close(100)
stop
end
```

You can compare `lit.tmp` and `big.tmp` files to see the difference of the byte order in these files.

Linux* Systems Only

On Linux* systems you can use the `od` utility to compare the files.

Example output

```
> od -t x4 lit.tmp
0000000 00000008 00000315 00000000 00000008
0000020 00000004 000001c8 00000004
0000034
> od -t x4 big.tmp
0000000 08000000 00000000 15030000 08000000
0000020 04000000 c8010000 04000000
0000034
```

You can see that the byte order is different in these files. If `info` and `od` are installed on your Linux system, enter `info od` at the prompt to get more information about the utility.

Symbol Visibility Attribute Options (Linux* and Mac OS* X)

Applications that do not require symbol preemption or position-independent code can obtain a performance benefit by taking advantage of the generic ABI visibility attributes.

Global Symbols and Visibility Attributes

A global symbol is a symbol that is visible outside the compilation unit in which it is declared (compilation unit is a single-source file with the associated include files). Each global symbol definition or reference in a compilation unit has a visibility attribute that controls how it may be referenced from outside the component in which it is defined.

The values for visibility are defined and described in the following topic:

- `-fvisibility` compiler option

NOTE. Visibility applies to both references and definitions. A symbol reference's visibility attribute is an assertion that the corresponding definition will have that visibility.

Symbol Preemption and Optimization

Sometimes programmers need to use some of the functions or data items from a shareable object, but at the same time, they need to replace other items with definitions of their own. For example, an application may need to use the standard run-time library shareable object, `libc.so`, but to use its own definitions of the heap management routines `malloc` and `free`.

NOTE. In this case it is important that calls to `malloc` and `free` within `libc.so` use the user's definition of the routines and not the definitions in `libc.so`. The user's definition should then override, or *`preempt`*, the definition within the shareable object.

This functionality of redefining the items in shareable objects is called symbol preemption. When the run-time loader loads a component, all symbols within the component that have default visibility are subject to preemption by symbols of the same name in components that are already loaded. Note that since the main program image is always loaded first, none of the symbols it defines will be preempted (redefined).

The possibility of symbol preemption inhibits many valuable compiler optimizations because symbols with default visibility are not bound to a memory address until run-time. For example, calls to a routine with default visibility cannot be inlined because the routine might be preempted if the compilation unit is linked into a shareable object. A preemptable data symbol cannot be accessed using GP-relative addressing because the name may be bound to a symbol in a different component; and the GP-relative address is not known at compile time.

Symbol preemption is a rarely used feature and has negative consequences for compiler optimization. For this reason, by default the compiler treats all global symbol definitions as non-preemptable (protected visibility). Global references to symbols defined in another compilation unit are assumed by default to be preemptable (default visibility). In those rare cases where all global definitions as well as references need to be preemptable, you can override this default.

Specifying Symbol Visibility Explicitly

The Intel® compiler has visibility attribute options that provide command-line control of the visibility attributes in addition to a source syntax to set the complete range of these attributes.

The options ensure immediate access to the feature without depending on header file modifications. The visibility options cause all global symbols to get the visibility specified by the option. There are two variety of options to specify symbol visibility explicitly.

Example
`-fvisibility=keyword` `-fvisibility-keyword=` *file*

The first form specifies the default visibility for global symbols. The second form specifies the visibility for symbols that are in a file (this form overrides the first form).

Specifying Visibility without the Symbol File

This option sets the visiblity for symbols not specified in a visibility list file and that do not have `VISIBILITY` attribute in their declaration. If no symbol file option is specified, all symbols will get the specified attribute. Command line example:

Example
`ifort -fvisibility=protected a.f`

You can set the default visibility for symbols using one of the following command line options:

Examples

`-fvisibility=extern`
`-fvisibility=default`
`-fvisibility=protected`
`-fvisibility=hidden`
`-fvisibility=internal`

Data Alignment Options

These options control how the Intel® compiler align data items. Refer to Compiler Options for more information on using these alignment-related options.

Linux* and Mac OS* X	Windows*	Description
`-align rec`*n*byte	`/align:rec`*n*byte	Specifies the alignment constraint for structures on *n*-byte boundaries.

Linux* and Mac OS* X	Windows*	Description
These front-end options changes alignment of variables in a common block. For more information, see the following topic:		For more information, see the following topic:
No equivalent	/Qsfalign IA-32 architecture Only	This option aligns stack for functions. For more information, see the following topic:

Part IV

Floating-point Operations

Topics:

- Overview: Floating-point Operations
- Floating-point Options Quick Reference
- Understanding Floating-point Operations
- Tuning Performance
- Handling Floating-point Exceptions
- Understanding IEEE Floating-point Operations

Overview: Floating-point Operations

35

This section introduces the floating-point support in the Intel® Fortran Compiler and provides information about using floating-point operations in your applications. The section also briefly describes the IEEE* Floating-Point Standard (IEEE 754).

The following table lists some possible starting points:

If you are trying to...	Then start with...
Understand the programming trade-offs in floating-point applications	Programming Trade-offs in Floating-Point Applications
Use the `-fp-model` (Linux* and Mac OS* X) or `/fp` (Windows*) option	Using the `-fp-model` or `/fp` Option
Set the flush-to-zero (FTZ) or denormals-are-zero (DAZ) flags	Setting the FTZ and DAZ Flags
Handle floating-point exceptions	Handling Floating-Point Exceptions
Tune the performance of floating-point applications for consistency	Overview: Tuning Performance of Floating-Point Applications
Learn about the IEEE Floating-Point Standard	Overview: Understanding IEEE Floating-Point Standard

Floating-point Options Quick Reference

36

The Intel® Compiler provides various options for you to optimize floating-point calculations with varying degrees of accuracy and predictability on different Intel architectures. This topic lists these compiler options and provides information about their supported architectures and operating systems.

IA-32, Intel® 64, and IA-64 architectures

Linux* and Mac OS* X	Windows*	Description
`-fp-model`	`/fp`	Specifies semantics used in floating-point calculations. Values are `precise`, `fast [=1/2]`, `strict`, `source`, `double`, `extended`, `[no-]except` (Linux* and MacOS* X) and `except[-]` (Windows*). • `-fp-model` compiler option
`-fp-speculation`	`/Qfp-speculation`	Specifies the speculation mode for floating-point operations. Values are `fast`, `safe`, `strict`, and `off`. • `-fp-speculation` compiler option
`-prec-div, -no-prec-div`	`/Qprec-div, /Qprec-div-`	Attempts to use slower but more accurate implementation of floating-point divide. Use this option to disable the divide optimizations in cases where it is important to maintain the full range and precision for floating-point division. Using this option results in greater accuracy with some loss of performance. Specifying `-no-prec-div` (Linux* and Mac OS* X) or `/Qprec-div-` (Windows*) enable the divide-to-reciprocal multiply optimization; these results are slightly less precise than full IEEE division results.

Linux* and Mac OS* X	Windows*	Description
		• -prec-div compiler option
-complex-limited-range	/Qcomplex-limited-range	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX. This can cause performance improvements in programs that use a lot of COMPLEX arithmetic. Values at the extremes of the exponent range might not compute correctly; for example, for single-precision floating-point operations, values >1.E20 or <1.E-20 will not compute correctly . • -complex-limited-range compiler option
-ftz	/Qftz	May flush denormal results to zero. The default behavior depends on the architecture. Refer to the following topic for details: • -ftz compiler option
-fpe, -fpe-all	/fpe, /fpe-all	By default, the Fortran compiler disables all floating-point exceptions; whether floating underflow defaults to gradual or abrupt is architecture-dependent. These options control which exceptions are enabled by the Fortran compiler. They also control whether floating-point underflow is gradual or abrupt. • -fpe compiler option • -fpe-all compiler option

IA-32 and Intel® 64 Architectures

Linux* and Mac OS* X	Windows*	Description
`-prec-sqrt`	`/Qprec-sqrt`	Improves the accuracy of square root implementations, but using this option may impact speed. • `-prec-sqrt` compiler option
`-pc`	`/Qpc`	Changes the floating point significand precision in the x87 control word. The application must use `PROGRAM` as the entry point, and you must compile the source file containing `PROGRAM` with this option. • `-pc` compiler option
`-rcd`	`/Qrcd`	Disables rounding mode changes for floating-point-to-integer conversions. • `-rcd` compiler option
`-fp-port`	`/Qfp-port`	Causes floating-point values to be rounded to the source precision at assignments and casts. • `-fp-port` compiler option
`-mp1`	`/Qprec`	This option rounds floating-point values to the precision specified in the source program prior to comparisons. It also implies `-prec-div` and `-prec-sqrt` (Linux and Mac OS X) or `/Qprec-div` and `/Qprec-sqrt` (Windows). • `-mp1` compiler option

IA-64 architecture only

Linux*	Windows*	Description
`-fma, -no-fma`	`/Qfma, /Qfma-`	Enables/disables the contraction of floating-point multiply and add/subtract operations into a single operation. • `-fma` compiler option • `-fp-model` compiler options
`-fp-relaxed`	`/Qfp-relaxed`	Enables use of faster but slightly less accurate code sequences for math functions, such as the `sqrt()` function and the divide operation. As compared to strict IEEE* precision, using this option slightly reduces the accuracy of floating-point calculations performed by these functions, usually limited to the least significant binary digit. • `-fp-relaxed` compiler option

Understanding Floating-point Operations

37

Programming Tradeoffs in Floating-point Applications

In general, the programming objectives for floating-point applications fall into the following categories:

- **Accuracy:** The application produces results that are close to the correct result.
- **Reproducibility and portability:** The application produces consistent results across different runs, different sets of build options, different compilers, different platforms, and different architectures.
- **Performance:** The application produces fast, efficient code.

Based on the goal of an application, you will need to make tradeoffs among these objectives. For example, if you are developing a 3D graphics engine, then performance may be the most important factor to consider, and reproducibility and accuracy may be your secondary concerns.

The Intel® Compiler provides appropriate compiler options, such as the `-fp-model` (Linux* and Mac OS* X operating systems) or `/fp` (Windows* operating system) option, that allow you to tune your applications based on specific objectives. The compiler optimizes and generates code differently when you specify different compiler options. Take the following code as an example:

```
REAL(4):: t0, t1, t2

...

t0=t1+t2+4.0+0.1
```

If you specify the `-fp-model extended` (Linux* and Mac OS* X) or `/fp:extended` (Windows*) option in favor of accuracy, the compiler generates the following assembly code:

```
fld       DWORD PTR _t1

fadd      DWORD PTR _t2

fadd      DWORD PTR _Cnst4.0

fadd      DWORD PTR _Cnst0.1

fstp      DWORD PTR _t0
```

The above code maximizes accuracy because it utilizes the highest mantissa precision available on the target platform. However, the code might suffer in performance due to the overhead of managing the x87 stack and it might yield results that cannot be reproduced on other platforms that do not have an equivalent extended precision type.

If you specify the `-fp-model source` (Linux* and Mac OS* X) or `/fp:source` (Windows*) option in favor of reproducibility and portability, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
addss     xmm0, DWORD PTR _Cnst4.0
addss     xmm0, DWORD PTR _Cnst0.1
movss     DWORD PTR _t0, xmm0
```

The above code maximizes portability by preserving the original order of the computation and by using the well-defined IEEE single-precision type for all computations. It is not as accurate as the previous implementation because the intermediate rounding error is greater compared to extended precision. And it is not the highest performance implementation because it does not take advantage of the opportunity to precompute `4.0 + 0.1`.

If you specify the `-fp-model fast` (Linux* and Mac OS* X) or `/fp:fast` (Windows*) option in favor of performance, the compiler generates the following assembly code:

```
movss     xmm0, DWORD PTR _Cnst4.1
addss     xmm0, DWORD PTR _t1
addss     xmm0, DWORD PTR _t2
movss     DWORD PTR _t0, xmm0
```

The above code maximizes performance by using Intel® SSE instructions and precomputing `4.0 + 0.1`. It is not as accurate as the first implementation, again due to greater intermediate rounding error. It will not provide reproducible results like the second implementation because it must reorder the addition in order to precompute `4.0 + 0.1`, and you cannot expect that all compilers, on all platforms, at all optimization levels will reorder the addition in the same way.

For most other applications, the considerations may be more complicated. You should select appropriate compiler options by carefully balancing your programming objectives and making tradeoffs among these objectives.

Floating-point Optimizations

Application performance is an important goal of the Intel® Compilers, even at default optimization levels. A number of optimizations involve transformations that might affect the floating-point behavior of the application, such as evaluation of constant expressions at compile time, hoisting

invariant expressions out of loops, or changes in the order of evaluation of expressions. These optimizations usually help the compiler to produce the most efficient code possible. However, the optimizations might be contrary to the floating-point requirements of the application.

Some optimizations are not consistent with strict interpretation of the ANSI or ISO standards for Fortran. Such optimizations can cause differences in rounding and small variations in floating-point results that may be more or less accurate than the ANSI-conformant result.

Intel Compilers provide the `-fp-model` (Linux* and Mac OS* X) or `/fp` (Windows*) option, which allows you to control the optimizations performed when you build an application. The option allows you to specify the compiler rules for:

- **Value safety:** Whether the compiler may perform transformations that could affect the result. For example, in the SAFE mode, the compiler won't transform `x/x` to 1.0 because the value of `x` at runtime might be a zero or a NaN . The UNSAFE mode is the default.
- **Floating-point expression evaluation:** How the compiler should handle the rounding of intermediate expressions.
- **Floating-point contractions:** Whether the compiler should generate fused multiply-add (FMA) instructions on processors based on the IA-64 architecture. When enabled, the compiler may generate FMA instructions for combining multiply and add operations; when disabled, the compiler must generate separate multiply and add instructions with intermediate rounding.
- **Floating-point environment access:** Whether the compiler must account for the possibility that the program might access the floating-point environment, either by changing the default floating-point control settings or by reading the floating-point status flags. This is disabled by default. You can use the `-fp-model:strict` (Linux and Mac OS X) `/fp:strict` (Windows) option to enable it.
- **Precise floating-point exceptions:** Whether the compiler should account for the possibility that floating-point operations might produce an exception. This is disabled by default. You can use `-fp-model:strict` (Linux and Mac OS X) or `/fp:strict` (Windows); or `-fp-model:except` (Linux and Mac OS X) or `/fp:except` (Windows) to enable it.

The following table describes the impact of different keywords of the option on compiler rules and optimizations:

Keyword	Value Safety	Floating-Point Expression Evaluation	Floating-Point Contractions	Floating-Point Environment Access	Precise Floating-Point Exceptions
`precise source`	Varies	Source Source	Yes	No	No
`strict`	Varies	Source	No	Yes	Yes

Keyword	Value Safety	Floating-Point Expression Evaluation	Floating-Point Contractions	Floating-Point Environment Access	Precise Floating-Point Exceptions
`fast=1` (default)	Unsafe	Unknown	Yes	No	No
`fast=2`	Unsafe	Unknown	Yes	No	No
`except`	Unaffected	Source	Unaffected	Unaffected	Yes
`except-`	Unaffected	Source	Unaffected	Unaffected	No

NOTE. It is illegal to specify the `except` keyword in an unsafe safety mode.

Based on the objectives of an application, you can choose to use different sets of compiler options and keywords to enable or disable certain optimizations, so that you can get the desired result.

See Also

- Understanding Floating-point Operations
- Using -fp-model (/fp) Option

Using the -fp-model (/fp) Option

The `-fp-model` (Linux* and Mac OS* X) or `/fp` (Windows*) option allows you to control the optimizations on floating-point data. You can use this option to tune the performance, level of accuracy, or result consistency for floating-point applications across platforms and optimization levels.

For applications that do not require support for denormalized numbers, the `-fp-model` or `/fp` option can be combined with the `-ftz` (Linux*and Mac OS* X) or `/Qftz` (Windows*) option to flush denormalized results to zero in order to obtain improved runtime performance on processors based on all Intel architectures (IA-32, Intel® 64, and IA-64 architectures).

You can use keywords to specify the semantics to be used. Possible values of the keywords are as follows:

Keyword	Description
`precise`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision.
`fast[=1|2]`	Enables more aggressive optimizations on floating-point data.
`strict`	Enables `precise` and `except`, disables contractions, and enables the property that allows modification of the floating-point environment.
`source`	Enables value-safe optimizations on floating-point data and rounds intermediate results to source-defined precision (same as precise keyword).
`[no-]except` (Linux* and Mac OS* X) or `except[-]` (Windows*)	Determines whether strict floating-point exception semantics are used.

The default value of the option is `-fp-model fast=1` or `/fp:fast=1`, which means that the compiler uses more aggressive optimizations on floating-point calculations.

NOTE. Using the default option keyword `-fp-model fast` or `/fp:fast`, you may get significant differences in your result depending on whether the compiler uses x87 or SSE2 instructions to implement floating-point operations. Results are more consistent when the other option keywords are used.

Several examples are provided to illustrate the usage of the keywords. These examples show:

- A small example of source code
  Note that the same source code is considered in all the included examples.
- The semantics that are used to interpret floating-point calculations in the source code
- One or more possible ways the compiler may interpret the source code
  Note that there are several ways the compiler may interpret the code; we show just some of these possibilities.

-fp-model fast or /fp:fast

Example source code:

```
REAL T0, T1, T2;

...

T0 = 4.0E + 0.1E + T1 + T2;
```

When this option is specified, the compiler applies the following semantics:

- Additions may be performed in any order
- Intermediate expressions may use `single`, `double`, or `extended double` precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, some possible ways the compiler may interpret the original code are given below:

```
REAL T0, T1, T2;


...

T0 = (T1 + T2) + 4.1E;

REAL T0, T1, T2;

...

T0 = (T1 + 4.1E) + T2;
```

-fp-model source or /fp:source

This setting is equivalent to `-fp-model precise` or `/fp:precise` on systems based on the Intel® 64 architecture.

Example source code:

```
REAL T0, T1, T2;

...

T0 = 4.0E + 0.1E + T1 + T2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order

- Intermediate expressions use the precision specified in the source code, that is, single-precision
- The constant addition may be pre-computed, assuming the default rounding mode

Using these semantics, a possible way the compiler may interpret the original code is shown below:

```
REAL T0, T1, T2;
...
T0 = ((4.1E + T1) + T2);
```

-fp-model strict or /fp:strict

Example source code:

```
REAL T0, T1, T2;
...
T0 = 4.0E + 0.1E + T1 + T2;
```

When this option is specified, the compiler applies the following semantics:

- Additions are performed in program order
- Intermediate expressions always use source precision in modes other than `fast`.
- The constant addition is not pre-computed because there is no way to tell what rounding mode will be active when the program runs.

Using these semantics, a possible way the compiler may interpret the original code is shown below:

```
REAL T0, T1, T2;
...
T0 = REAL ((((REAL)4.0E + (REAL)0.1E) + (REAL)T1) + (REAL)T2);
```

See Also

- Understanding Floating-point Operations
- `-fp-model` compiler option

Denormal Numbers

A normalized number is a number for which both the exponent (including bias) and the most significant bit of the mantissa are non-zero. For such numbers, all the bits of the mantissa contribute to the precision of the representation.

The smallest normalized single-precision floating-point number greater than zero is about 1.1754943^{-38}. Smaller numbers are possible, but those numbers must be represented with a zero exponent and a mantissa whose leading bit(s) are zero, which leads to a loss of precision. These numbers are called denormalized numbers or denormals (newer specifications refer to these as subnormal numbers).

Denormal computations use hardware and/or operating system resourcesto handle denormals; these can cost hundreds of clock cycles.

- Denormal computations take much longer to calculate on processors based on IA-32 and Intel® 64 architectures than normal computations.
- Denormals are computed in software on processors based on the IA-64 architecture. The computation usually requires hundreds of clock cycles that results in excessive kernel time.

There are several ways to avoid denormals and increase the performance of your application:

- Scale the values into the normalized range.
- Use a higher precision data type with a larger range.
- Flush denormals to zero.

See Also

- Understanding Floating-point Operations

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture

Institute of Electrical and Electronics Engineers, Inc*. (IEEE) web site for information about the current floating-point standards and recommendations

Floating-point Environment

The floating-point environment is a collection of registers that control the behavior of the floating-point machine instructions and indicate the current floating-point status. The floating-point environment can include rounding mode controls, exception masks, flush-to-zero (FTZ) controls, exception status flags, and other floating-point related features.

For example, on IA-32 and Intel® 64 architectures, bit 15 of the MXCSR register enables the flush-to-zero mode, which controls the masked response to an single-instruction multiple-data (SIMD) floating-point underflow condition.

The floating-point environment affects most floating-point operations; therefore, correct configuration to meet your specific needs is important. For example, the exception mask bits define which exceptional conditions will be raised as exceptions by the processor. In general, the default floating-point environment is set by the operating system. You don't need to configure the floating-point environment unless the default floating-point environment does not suit your needs.

There are several methods available if you want to modify the default floating-point environment. For example, you can use inline assembly, compiler built-in functions, library functions, or command line options.

Changing the default floating-point environment affects runtime results only. This does not affect any calculations that are pre-computed at compile time.

See Also

- Understanding Floating-point Operations

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Intel® Itanium® Architecture Software Developer's Manual, Volume 1: Application Architecture

Setting the FTZ and DAZ Flags

In Intel® processors, the flush-to-zero (FTZ) and denormals-are-zero (DAZ) flags in the `MXCSR` register are used to control floating-point calculations. The Intel® Streaming SIMD (Single Instruction Multiple Data) Extensions (Intel® SSE) and the Intel® SSE 2 instructions, including scalar and vector instructions, benefit from enabling the FTZ and DAZ flags respectively. Floating-point computations using these Intel® SSE instructions are accelerated when the FTZ and DAZ flags are enabled and thus the performance of the application improves.

You can use the `-ftz` (Linux* and Mac OS* X) or `/Qftz` (Windows*) option to flush denormal results to zero when the application is in the gradual underflow mode. This option may improve performance if the denormal values are not critical to your application's behavior. The `-ftz` and `/Qftz` options, when applied to the main program, set the FTZ and the DAZ hardware flags. The `-no-ftz` and `/Qftz-` options leave the flags as they are.

The following table describes how the compiler processes denormal values based on the status of the FTZ and DAZ flags:

Flag	When set to ON, the compiler...	When set to OFF, the compiler...	Supported on
FTZ (flush-to-zero)	Sets denormal results from floating-point calculations to zero	Does not change the denormal results	IA-64, Intel® 64 architectures, and some IA-32 architectures
DAZ (denormals-are-zero)	Treats denormal values used as input to floating-point instructions as zero	Does not change the denormal instruction inputs	Intel® 64 architecture and some IA-32 architecture

- FTZ and DAZ are not supported on all IA-32 architectures. The FTZ flag is supported only on IA-32 architectures that support Intel® SSE instructions.
- On systems based on the IA-64 architecture, FTZ always works, while on systems based on the IA-32 and Intel® 64 architectures, FTZ only applies to Intel® SSE instructions. Hence, if your application happens to generate denormals using x87 instructions, FTZ does not apply.
- DAZ and FTZ flags are not compatible with IEEE Standard 754, so you should only consider enabling them when strict compliance to the IEEE standard is not required.

Options `-ftz` and `/Qftz` are performance options. Setting these options does not guarantee that all denormals in a program are flushed to zero. They only cause denormals generated at run-time to be flushed to zero.

On Intel®64 and IA-32 systems, the compiler, by default, inserts code into the main routine to set the FTZ and DAZ flags. When `-ftz` or `/Qftz` option is used on IA-32 systems with the option `-msse2` or `/arch:sse2`, the compiler will insert code to conditionally set FTZ/DAZ flags based on a run-time processor check. The `-no-ftz` (Linux* and Mac OS* X) or `/Qftz-` (Windows) will prevent the compiler from inserting any code that might set FTZ or DAZ flags.

When `-ftz` or `/Qftz` is used in combination with an Intel® SSE-enabling option on systems based on the IA-32 architecture (for example, `-msse2` or `/arch:sse2`), the compiler will insert code in the main routine to set FTZ and DAZ. When `-ftz` or `/Qftz` is used without such an option, the compiler will insert code to conditionally set FTZ/DAZ based on a run-time processor check. `-no-ftz` (Linux and Mac OS X) or `/Qftz-` (Windows) will prevent the compiler from inserting any code that might set FTZ or DAZ.

The `-ftz` or `/Qftz` option only has an effect when the main program is being compiled. It sets the FTZ/DAZ mode for the process. The initial thread and any threads subsequently created by that process will operate in the FTZ/DAZ mode.

On systems based on the IA-64 architecture, optimization option O3 sets `-ftz` and `/Qftz`; optimization option O2 sets `-no-ftz` (Linux) and `/Qftz-` (Windows). On systems based on the IA-32 and Intel® 64 architectures, every optimization option O level, except O0, sets `-ftz` and `/Qftz`.

If this option produces undesirable results of the numerical behavior of your program, you can turn the FTZ/DAZ mode off by using `-no-ftz` or `/Qftz-` in the command line while still benefiting from the O3 optimizations.

For some non-Intel processors, the flags can be set manually by calling the following Intel Fortran intrinsic:

Example
`RESULT = FOR_SET_FPE (FOR_M_ABRUPT_UND)`

See Also

- Understanding Floating-point Operations

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Checking the Floating-point Stack State

On systems based on the IA-32 architectures, when an application calls a function that returns a floating-point value, the returned floating-point value is supposed to be on the top of the floating-point stack. If the return value is not used, the compiler must pop the value off of the floating-point stack in order to keep the floating-point stack in the correct state.

On systems based on Intel(R) 64 architectures, floating-point values are usually returned in the `xmm0` register. The floating-point stack is used only when the return value is an internal 80-bit floating-point data type on Linux* and Mac OS* X systems.

If the application calls a function without defining or incorrectly defining the function's prototype, the compiler cannot determine if the function must return a floating-point value. Consequently, the return value is not popped off the floating-point stack if it is not used. This can cause the floating-point stack to overflow.

The overflow of the stack results in two undesirable situations:

- A `NaN` value gets involved in the floating-point calculations
- The program results become unpredictable; the point where the program starts making errors can be arbitrarily far away from the point of the actual error.

For systems based on the IA-32 and Intel® 64 architectures, the `-fp-stack-check` (Linux* and Mac OS* X) or `/Qfp-stack-check` (Windows*) option checks whether a program makes a correct call to a function that should return a floating-point value. If an incorrect call is detected, the option places a code that marks the incorrect call in the program. The `-fp-stack-check` (Linux* and Mac OS* X) or `/Qfp-stack-check` (Windows*) option marks the incorrect call and makes it easy to find the error.

NOTE. The `-fp-stack-check` (Linux* and Mac OS* X) and the `/Qfp-stack-check` (Windows*) option causes significant code generation after every function/subroutine call to ensure that the floating-point stack is maintained in the correct state. Therefore, using this option slows down the program being compiled. Use the option only as a debugging aid to find floating point stack underflow/overflow problems, which can be otherwise hard to find.

See Also

- Understanding Floating-point Operations
- -fp-stack-check, /Qfp-stack-check option

Tuning Performance

38

Overview: Tuning Performance

This section describes several programming guidelines that can help you improve the performance of a floating-point applications:

- Avoid exact floating-point comparisons
- Avoid exceeding representable ranges during computation; handling these cases can have a performance impact.
- Use REAL variables in single precision format unless the extra precision obtained through `DOUBLE` or `REAL*8` is required because a larger precision formation will also increase memory size and bandwidth requirements. See Using Efficient Data Types section.
- Reduce the impact of denormal exceptions for all supported architectures.
- Avoid mixed data type arithmetic expressions.

Avoiding Exact Floating-point Comparison

It is unsafe for applications to rely on exact floating-point comparisons. Slight variations in rounding can change the outcome of such comparisons, leading to non-convergence or other unexpected behavior.

Tests for equality of floating-point quantities should be made within some tolerance related to the expected precision of the calculation, for example, by using the Fortran intrinsic function EPSILON. The following examples demonstrate the concept:

Example

```
if (foo() == 2.0)
```

Where `foo()` may be as close to 2.0 as can be imagined without actually exactly matching 2.0. You can improve the behavior of such codes by using inexact floating-point comparisons or fuzzy comparisons to test a value to within a certain tolerance, as shown below:

Example

```
epsilon = 1E-8;
if (abs(foo() - 2.0) <= epsilon)
```

Handling Floating-point Array Operations in a Loop Body

Following the guidelines below will help autovectorization of the loop.

- Statements within the loop body may contain float or double operations (typically on arrays). The following arithmetic operations are supported: addition, subtraction, multiplication, division, negation, square root, MAX, MIN, and mathematical functions such as SIN and COS.
- Writing to a single-precision scalar/array and a double scalar/array within the same loop decreases the chance of autovectorization due to the differences in the vector length (that is, the number of elements in the vector register) between float and double types. If autovectorization fails, try to avoid using mixed data types.

See Also

- Tuning Performance
- Programming Guidelines for Vectorization

Reducing the Impact of Denormal Exceptions

Denormalized floating-point values are those that are too small to be represented in the normal manner; that is, the mantissa cannot be left-justified. Denormal values require hardware or operating system interventions to handle the computation, so floating-point computations that result in denormal values may have an adverse impact on performance.

There are several ways to handle denormals to increase the performance of your application:

- Scale the values into the normalized range
- Use a higher precision data type with a larger range
- Flush denormals to zero

For example, you can translate them to normalized numbers by multiplying them using a large scalar number, doing the remaining computations in the normal space, then scaling back down to the denormal range. Consider using this method when the small denormal values benefit the program design.

If you change the declaration of a variable you might also need to change the libraries you call to use the variable Another strategy that might result in increased performance is to increase the amount of precision of intermediate values using the `-fp-model [double|extended]` option. However, this strategy might not eliminate all denormal exceptions, so you must experiment with the performance of your application.

If you change the type declaration of a variable, you might also need to change associated library calls, unless these are generic Another strategy that might result in increased performance is to increase the amount of precision of intermediate values using the `-fp-model [double|extended]` option. However, this strategy might not eliminate all denormal exceptions, so you must experiment with the performance of your application. You should verify that the gain in performance from eliminating denormals is greater than the overhead of using a data type with higher precision and greater dynamic range.

Finally, in many cases denormal numbers can be treated safely as zero without adverse effects on program results. Depending on the target architecture, use flush-to-zero (FTZ) options.

IA-32 and Intel® 64 Architectures

These architectures take advantage of the FTZ (flush-to-zero) and DAZ (denormals-are-zero) capabilities of Intel® Streaming SIMD Extensions (Intel® SSE) instructions.

On Intel®64 and IA-32-based systems, the compiler, by default, inserts code into the main routine to enable FTZ and DAZ at optimization levels higher than `-O0`. To enable FTZ and DAZ at `-O0`, compile the source file containing PROGRAM using `-ftz` or `/Qftz` option. When `-ftz` or `/Qftz` option is used on IA-32-based systems with the option `-mia32` or `/arch:IA32`, the compiler inserts code to conditionally enable FTZ and DAZ flags based on a run-time processor check.

NOTE. After using flush-to-zero, ensure that your program still gives correct results when treating denormalized values as zero.

IA-64 Architecture

Enable the FTZ mode by using the `-ftz` (Linux and Mac OS X) or `/Qftz` (Windows) option on the source file containing PROGRAM. The `-O3` (Linux and Mac OS X) or `/O3` (Windows) option automatically enables `-ftz` or `/Qftz`.

NOTE. After using flush-to-zero, ensure that your program still gives correct results when treating denormalized values as zero.

See Also

- Tuning Performance
- Setting the FTZ and DAZ Flags

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture

Avoiding Mixed Data Type Arithmetic Expressions

Avoid mixing integer and floating-point (REAL) data in the same computation. Expressing all numbers in a floating-point arithmetic expression (assignment statement) as floating-point values eliminates the need to convert data between fixed and floating-point formats. Expressing all numbers in an integer arithmetic expression as integer values also achieves this. This improves run-time performance.

For example, assuming that `I` and `J` are both INTEGER variables, expressing a constant number (2.) as an integer value (2) eliminates the need to convert the data. The following examples demonstrate inefficient and efficient code.

Examples

Example 1: Inefficient Code

```
INTEGER I, J
I = J / 2.
```

Example 2: Efficient Code

```
INTEGER I, J
I = J / 2
```

Special Considerations for Auto-Vectorization of the Innermost Loops

Auto-vectorization of an innermost loop packs multiple data elements from consecutive loop iterations into a vector register, each of which is 128-bit in size.

Consider a loop that uses different sized data, for example, REAL and DOUBLE PRECISION. For REAL data, the compiler tries to pack data elements from four (4) consecutive iterations (32 bits x 4 = 128 bits). For DOUBLE PRECISION data, the compiler tries to pack data elements from two (2) consecutive iterations (64 bits x 2 = 128 bits). Because of the mismatched number of iterations, the compiler sometimes fails to perform auto-vectorization of the loop, after trying to automatically remedy the situation.

If your attempt to auto-vectorize an innermost loop fails, it is a good practice to try using the same sized data. INTEGER and REAL are considered same sized data since both are 32-bit in size.

Examples

Example 1: Non-autovectorizable code

```
DOUBLE PRECISION A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=D(I)
   C(I)=B(I)
ENDDO
```

Example 2: Auto-vectorizable after automatic distribution into two loops

```
DOUBLE PRECISION A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=B(I)
   C(I)=D(I)
ENDDO
```

Example 3: Auto-vectorizable as one loop

```
REAL A(N), B(N)
REAL C(N), D(N)
DO I=1, N
   A(I)=B(I)
   C(I)=D(I)
ENDDO
```

Using Efficient Data Types

In cases where more than one data type can be used for a variable, consider selecting the data types based on the following hierarchy, listed from most to least efficient:

- Integer
- Single-precision real, expressed explicitly as `REAL`, `REAL (KIND=4)`, or `REAL*4`
- Double-precision real, expressed explicitly as `DOUBLE PRECISION`, `REAL (KIND=8)`, or `REAL*8`
- Extended-precision real, expressed explicitly as `REAL (KIND=16)` or `REAL*16`

However, keep in mind that in an arithmetic expression, you should avoid mixing integer and floating-point data.

Handling Floating-point Exceptions

39

Overview: Controlling Floating-point Exceptions

When developing applications, you may need to control the exceptions that can occur during the run-time processing of floating-point numbers. These exceptions can be categorized into specific types: overflow, divide-by-zero, underflow, and invalid operations.

Overflow

Overflow is signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result. The result computed is rounding mode specific:

- Round-to-nearest (default): +/- Infinity in specified precision
- Round-to-zero: +/- Maximum Number in specified precision
- Round-to-+Infinity: +Infinity or -(Maximum Positive Number) in specified precision
- Round-to--Infinity: (Maximum Positive Number) or -Infinity in specified precision

For example, in round-to-nearest mode 1.E30 * 1.E30 overflows the single-precision floating-point range and results in a +Infinity; -1.E30 * 1.E30 results in a -Infinity.

Divide-by-zero

Divide-by-zero is signaled when the divisor is zero and the dividend is a finite nonzero number. The computed result is a correctly signed Infinity.
For example, 2.0E0/+0.0 produces a divide-by-zero exception and results in a +Infinity; -2.0E0/+0.0 produces a divide-by-zero exception and results in a -Infinity.

Underflow

Underflow occurs when a computed result (of an add, subtract, multiply, divide, or math function call) falls beyond the minimum range in magnitude of normalized numbers of the floating-point data type. Each floating-point type (32-, 64-, and 128-bit) has a denormalized range where very small numbers can be represented with some loss of precision. This is called gradual underflow. For example, the lower bound for normalized single-precision floating-point is approximately 1.E-38,

while the lower bound for denormalized single-precision floating-point is approximately 1.E-45. Results falling below the lower bound of the denormalized range simply become zero. 1.E-30 / 1.E10 underflows the normalized range but not the denormalized range so the result is the denormal value 1.E-40. 1.E-30 / 1.E30 underflows the entire range and the result is zero.

Invalid operation

Invalid occurs when operands to the basic floating-point operations or math function inputs produce an undefined (QNaN) result. Some examples include:

- SNaN operand in any floating-point operation or math function call
- Division of zeroes: (+/-0.0)/(+/-0.0)
- Sum of Infinities having different signs: Infinity + (-Infinity)
- Difference of Infinities having the same sign: (+/-Infinity) - (+/-Infinity)
- Product of signed Infinities with zero: (+/-Inf) * 0
- Math Function Domain Errors: log(negative), sqrt(negative), asin(|x|>1)

With the Intel® Compiler, you can use the `-fpe` (Linux* and Mac OS* X) or `/fpe` (Windows*) option to control floating-point exceptions.

Handling Floating-point Exceptions

If a floating-point exception is disabled (its bit is set to 1 with `SETCONTROLFPQQ` (x87 arithmetic only), it will not generate an interrupt signal if it occurs. The floating-point process may return an appropriate special value (for example, NaN or signed infinity) or may return an acceptable value (for example, in the case of a denormal operand), and the program will continue. If a floating-point exception is enabled (its bit is set to 0), it will generate an interrupt signal (software interrupt) if it occurs.

The following table lists the floating-point exception signals:

Parameter Name	Value in Hex	Description
`FPE$INVALID`	`#81`	Invalid result
`FPE$DENORMAL`	`#82`	Denormal operand
`FPE$ZERODIVIDE`	`#83`	Divide by zero
`FPE$OVERFLOW`	`#84`	Overflow

Parameter Name	Value in Hex	Description
FPE$UNDERFLOW	#85	Underflow
FPE$INEXACT	#86	Inexact precision

If a floating-point exception interrupt occurs and you do not have an exception handling routine, the run-time system will respond to the interrupt according to the behavior selected by the compiler option /fpe. Remember, interrupts only occur if an exception is enabled (set to 0).

If you do not want the default system exception handling, you need to write your own interrupt handling routine:

- Write a function that performs whatever special behavior you require on the interrupt.
- Register that function as the procedure to be called on that interrupt with SIGNALQQ.

Note that your interrupt handling routine must use the cDEC$ ATTRIBUTES option C.

The drawback of writing your own routine is that your exception-handling routine cannot return to the process that caused the exception. This is because when your exception-handling routine is called, the floating-point processor is in an error condition, and if your routine returns, the processor is in the same state, which will cause a system termination. Your exception-handling routine can therefore either branch to another separate program unit or exit (after saving your program state and printing an appropriate message). You cannot return to a different statement in the program unit that caused the exception-handling routine, because a global GOTO does not exist, and you cannot reset the status word in the floating-point processor.

If you need to know when exceptions occur and also must continue if they do, you must disable exceptions so they do not cause an interrupt, then poll the floating-point status word at intervals with GETSTATUSFPQQ (IA-32 architecture only) to see if any exceptions occurred. To clear the status word flags, call the CLEARSTATUSFPQQ (IA-32 architecture only) routine.

Polling the floating-point status word at intervals creates processing overhead for your program. In general, you will want to allow the program to terminate if there is an exception. An example of an exception-handling routine follows. The comments at the beginning of the SIGTEST.F90 file describe how to compile this example.

```
! SIGTEST.F90
!Establish the name of the exception handler as the
! function to be invoked if an exception happens.
! The exception handler hand_fpe is attached below.
  USE IFPORT
     INTERFACE
       FUNCTION hand_fpe (sigid, except)
          !DEC$ ATTRIBUTES C :: hand_fpe
          INTEGER(4) hand_fpe
          INTEGER(2) sigid, except
       END FUNCTION
     END INTERFACE
INTEGER(4) iret
REAL(4) r1, r2
r1 = 0.0
iret = SIGNALQQ(SIG$FPE, hand_fpe)
WRITE(*,*) 'Set exception handler. Return = ', iret
! Cause divide-by-zero exception
r1 = 0.0
r2 = 3/r1
END
! Exception handler routine hand_fpe
  FUNCTION hand_fpe (signum, excnum)
    !DEC$ ATTRIBUTES C :: hand_fpe
    USE IFPORT
    INTEGER(2) signum, excnum
```

```
  WRITE(*,*) 'In signal handler for SIG$FPE'
  WRITE(*,*) 'signum = ', signum
  WRITE(*,*) 'exception = ', excnum
  SELECT CASE(excnum)
    CASE( FPE$INVALID )
      STOP ' Floating point exception: Invalid number'
    CASE( FPE$DENORMAL )
      STOP ' Floating point exception: Denormalized number'
    CASE( FPE$ZERODIVIDE )
      STOP ' Floating point exception: Zero divide'
    CASE( FPE$OVERFLOW )
      STOP ' Floating point exception: Overflow'
    CASE( FPE$UNDERFLOW )
      STOP ' Floating point exception: Underflow'
    CASE( FPE$INEXACT )
      STOP ' Floating point exception: Inexact precision'
    CASE DEFAULT
      STOP ' Floating point exception: Non-IEEE type'
  END SELECT
  hand_fpe = 1
END
```

File fordef.for and Its Usage

The parameter file `fordef.for` contains symbols and INTEGER*4 values corresponding to the classes of floating-point representations. Some of these classes are exceptional ones such as bit patterns that represent positive denormalized numbers.

With this file of symbols and with the `FP_CLASS` intrinsic function, you have the flexibility of identifying exceptional numbers so that, for example, you can replace positive and negative denormalized numbers with true zero.

The following is a simple example of identifying floating-point bit representations:

```
include 'fordef.for'

real*4 a

integer*4 class_of_bits

a = 57.0

class_of_bits = fp_class(a)

if ( class_of_bits .eq. for_k_fp_pos_norm  .or.   &

     class_of_bits .eq. for_k_fp_neg_norm       ) then

   print *, a, ' is a non-zero and non-exceptional value'

else

   print *, a, ' is zero or an exceptional value'

end if

end
```

In this example, the symbol `for_k_fp_pos_norm` in the file `fordef.for` plus the REAL*4 value 57.0 to the `FP_CLASS` intrinsic function results in the execution of the first print statement.

The table below explains the symbols in the file `fordef.for` and their corresponding floating-point representations.

Symbols in `fordef.for`

Symbol Name	Class of Floating-Point Bit Representation
FOR_K_FP_SNAN	Signaling NaN
FOR_K_FP_QNAN	Quiet NaN
FOR_K_FP_POS_INF	Positive infinity
FOR_K_FP_NEG_INF	Negative infinity
FOR_K_FP_POS_NORM	Positive normalized finite number
FOR_K_FP_NEG_NORM	Negative normalized finite number
FOR_K_FP_POS_DENORM	Positive denormalized number
FOR_K_FP_NEG_DENORM	Negative denormalized number

Symbol Name	Class of Floating-Point Bit Representation
FOR_K_FP_POS_ZERO	Positive zero
FOR_K_FP_NEG_ZERO	Negative zero

Another example of using file `fordef.for` and intrinsic function FP_CLASS follows. The goals of this program are to quickly read any 32-bit pattern into a REAL*4 number from an unformatted file with no exception reporting and to replace denormalized numbers with true zero:

```
include 'fordef.for'

real*4 a(100)

integer*4 class_of_bits

! open an unformatted file as unit 1

!     ...

read (1) a

do i = 1, 100

  class_of_bits = fp_class(a(i))

  if ( class_of_bits .eq. for_k_fp_pos_denorm  .or.   &

       class_of_bits .eq. for_k_fp_neg_denorm       ) then

    a(i) = 0.0

  end if

end do

close (1)

end
```

You can compile this program with any value of `-fpe`*n* (Linux* and Mac OS* X) or `/fpe:`*n* (Windows*). Intrinsic function FP_CLASS helps to find and replace denormalized numbers with zeroes before the program can attempt to perform calculations on the denormalized numbers.

On the other hand, if this program did not replace denormalized numbers read from unit 1 with zeroes and the program was compiled with `-fpe0` or `/fpe:0`, then the first attempted calculation on a denormalized number would result in a floating-point exception. If you compile with `/fpe:0` flush-to-zero is enabled. If the resulting calculation creates a divide-by-zero, overflow, or invalid operation, then the application should abort with a floating-point exception. Otherwise, a program using the data will run to completion, perhaps faster and with different answers.

File `fordef.for` and intrinsic function `FP_CLASS` can work together to identify NaNs. A variation of the previous example would contain the symbols `for_k_fp_snan` and `for_k_fp_qnan` in the IF statement. A faster way to do this is based on the intrinsic ISNAN function. One modification of the previous example, using ISNAN, follows:

```
! The ISNAN function does not need file fordef.for

real*4 a(100)

! open an unformatted file as unit 1

!     ...

read (1) a

do i = 1, 100

  if ( isnan (a(i)) ) then

    print *, 'Element ', i, ' contains a NaN'

  end if

end do

close (1)

end
```

You can compile this program with any value of `-fpe`*n* or `/fpe:`*n*.

Setting and Retrieving Floating-point Status and Control Words (IA-32)

Overview: Setting and Retrieving Floating-point Status and Control Word

The FPU (floating-point unit) on systems based on the IA-32 architecture contains eight floating-point registers the system uses for numeric calculations, status and control words, and error pointers. You normally need to consider only the status and control words, and then only when customizing your floating-point environment.

The FPU status and control words correspond to 16-bit registers whose bits hold the value of a state of the FPU or control its operation. Intel Fortran defines a set of symbolic constants to set and reset the proper bits in the status and control words.

NOTE. The symbolic constants and the library routines used to read and write the control and status registers only affect the x87 control and status registers. They do not affect the MXCSR register (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions).

They do not affect the MXCSR (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions). For example:

```
USE IFPORT
INTEGER(2) status, control, controlo, mask_all_traps
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!     Save old control word
controlo = control
!     Clear the rounding control flags
control = IAND(control,NOT(FPCW$MCW_RC))
!     Set new control to round up
control = IOR(control,FPCW$UP)
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
! Demonstrate setting and clearing exception mask flags
mask_all_traps = FPCW$INVALID + FPCW$DENORMAL + &
    FPCW$ZERODIVIDE + FPCW$OVERFLOW + &
FPCW$UNDERFLOW + FPCW$INEXACT
!     Clear the exception mask flags
control = IAND(control,NOT(FPCW$MCW_EM))
!     Set new exception mask to disallow overflow
!     (i.e., enable overflow traps)
! but allow (i.e., mask) all other exception conditions.
control = IOR(control,IEOR(mask_all_traps,FPCW$OVERFLOW))
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
```

```
END
```

The status and control symbolic constants (such as FPCW$OVERFLOW and FPCW$CHOP in the preceding example) are defined as INTEGER(2) parameters in the module IFORT.F90 in the `...\INCLUDE` folder. The status and control words are logical combinations (such as with .AND.) of different parameters for different FPU options.

The name of a symbolic constant takes the general form *name*$*option*. The prefix *name* is one of the following:

Prefixes for Parameter Flags

name	Meaning
`FPSW`	Floating-point status word
`FPCW`	Floating-point control word
`SIG`	Signal
`FPE`	Floating-point exception
`MTH`	Math function

The suffix *option* is one of the options available for that *name*. The parameter *name*$*option* corresponds either to a status or control option (for example, FPSW$ZERODIVIDE, a status word parameter that shows whether a zero-divide exception has occurred or not) or *name*$*option* corresponds to a mask, which sets all symbolic constants to 1 for all the options of *name*. You can use the masks in logical functions (such as `IAND`, `IOR`, and `NOT`) to set or to clear all options for the specified *name*. The following sections define the *option*s and illustrate their use with examples.

You can control the floating-point processor options (on systems based on the IA-32 architecture) and find out its status with the run-time library routines GETSTATUSFPQQ (IA-32 architecture only), GETCONTROLFPQQ (IA-32 architecture only), and SETCONTROLFPQQ (IA-32 architecture only). Examples of using these routines also appear in the following sections.

See Also

- Setting and Retrieving Floating-point Status and Control Words (IA-32)
- Understanding Floating-Point Control Word (IA-32 architecture only)

Understanding Floating-point Status Word

On systems based on the IA-32 architecture, the FPU status word includes bits that show the floating-point exception state of the processor. The status word parameters describe six exceptions: invalid result, denormalized operand, zero divide, overflow, underflow and inexact precision. These are described in the section, Loss of Precision Errors. When one of the bits is set to 1, it means a past floating-point operation produced that exception type. (Intel Fortran initially clears all status bits. It does not reset the status bits before performing additional floating-point operations after an exception occurs. The status bits accumulate.)

The following table shows the floating-point exception status parameters:

Parameter Name	Value in Hex	Description
FPSW$MSW_EM	#003F	Status Mask (set all bits to 1)
FPSW$INVALID	#0001	An invalid result occurred
FPSW$DENORMAL	#0002	A denormal operand occurred
FPSW$ZERODIVIDE	#0004	A divide by zero occurred
FPSW$OVERFLOW	#0008	An overflow occurred
FPSW$UNDERFLOW	#0010	>An underflow occurred
FPSW$INEXACT	#0020	Inexact precision occurred

You can find out which exceptions have occurred by retrieving the status word and comparing it to the exception parameters. For example:

```
USE IFPORT
INTEGER(2) status
CALL GETSTATUSFPQQ(status)
IF (IAND (status, FPSW$INEXACT) > 0) THEN
    WRITE (*, *) "Inexact precision has occurred"
ELSE IF (IAND (status, FPSW$DENORMAL) > 0) THEN
    WRITE (*, *) "Denormal occurred"
END IF
```

To clear the status word flags, call the `CLEARSTATUSFPQQ` (IA-32 architecture only) routine.

NOTE. The GETSTAUSFPQQ and CLEARSTATUSFPQQ routines only affect the x87 status register. They do not affect the MXCSR register (the control and status register for the Intel® SSE and Intel® SSE2 instructions).

Floating-point Control Word Overview

On systems based on the IA-32 architecture, the FPU control word includes bits that control the FPU's precision, rounding mode, and whether exceptions generate signals if they occur. You can read the control word value with GETCONTROLFPQQ (IA-32 architecture only) to find out the current control settings, and you can change the control word with SETCONTROLFPQQ (IA-32 architecture only).

NOTE. The GETCONTROLFPQQ and SETCONTROLFPQQ routines only affect the x87 status register. They do not affect the MXCSR register (the control and status register for the SSE and SSE2 instructions).

Each bit in the floating-point control word corresponds to a mode of the floating-point math processor. The IFORT.F90 module file in the ...\INCLUDE folder contains the INTEGER(2) parameters defined for the control word, as shown in the following table:

Parameter Name	Value in Hex	Description
FPCW$MCW_PC	#0300	Precision control mask
FPCW$64	#0300	64-bit precision
FPCW$53	#0200	53-bit precision
FPCW$24	#0000	24-bit precision
FPCW$MCW_RC	#0C00	Rounding control mask
FPCW$CHOP	#0C00	Truncate
FPCW$UP	#0800	Round up
FPCW$DOWN	#0400	Round down
FPCW$NEAR	#0000	Round to nearest

Parameter Name	Value in Hex	Description
FPCW$MCW_EM	#003F	Exception mask
FPCW$INVALID	#0001	Allow invalid numbers
>FPCW$DENORMAL	#0002	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	#0004	Allow divide by zero
FPCW$OVERFLOW	#0008	Allow overflow
FPCW$UNDERFLOW	#0010	Allow underflow
FPCW$INEXACT	#0020	Allow inexact precision

The control word defaults are:

- 53-bit precision
- Round to nearest (rounding mode)
- The denormal, underflow, overflow, divide-by-zero, invalid, and inexact precision exceptions are disabled (do not generate an exception). To change exception handling, you can use the `-fpe` (Linux* and Mac OS* X) or the `/fpe` (Windows*) compiler option or the `FOR_SET_FPE` routine.

Using Exception, Precision, and Rounding Parameters

This topic describes the exception, precision, and rounding parameters that you can use for the control word.

Exception Parameters

An exception is disabled if its bit is set to 1 and enabled if its bit is cleared to 0. If an exception is disabled (exceptions can be disabled by setting the flags to 1 with SETCONTROLFPQQ [IA-32 architecture only]), it will not generate an interrupt signal if it occurs. The floating-point process will return an appropriate special value (for example, NaN or signed infinity), but the program continues. You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ (IA-32 architecture only).

If errors on floating-point exceptions are enabled (by clearing the flags to 0 with SETCONTROLFPQQ [IA-32 architecture only]), the operating system generates an interrupt when the exception occurs. By default these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

You should remember not to clear all existing settings when changing one. The values you want to change should be combined with the existing control word in an inclusive-OR operation (IOR) if you do not want to reset all options. For example:

```
USE IFPORT
INTEGER(2) control, newcontrol
CALL GETCONTROLFPQQ(control)
newcontrol = IOR(control,FPCW$INVALID)
! Invalid exception set (disabled).
CALL SETCONTROLFPQQ(newcontrol)
```

NOTE. The GETCONTROLFPQQ, SETCONTROLFPQQ, and GETSTATUSFPQQ routines only affect the x87 status register. They do not affect the MXCSR register (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions).

Precision Parameters

On systems based on the IA-32 architecture, the precision bits control the precision to which the FPU rounds floating-point numbers. For example:

```
USE IFPORT
INTEGER(2) control, holdcontrol, newcontrol
CALL GETCONTROLFPQQ(control)
! Clear any existing precision flags.
holdcontrol = IAND(control, NOT(FPCW$MCW_PC))
newcontrol = IOR(holdcontrol, FPCW$64)
! Set precision to 64 bits.
CALL SETCONTROLFPQQ(newcontrol)
```

The precision options are mutually exclusive. If you set more than one, you may get an invalid mode or a mode other than the one you want. Therefore, you should clear the precision bits before setting a new precision mode.

NOTE. The `GETCONTROLFPQQ` and `SETCONTROLFPQQ` routines only affect the x87 status register. They do not affect the `MXCSR` register (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions).

Rounding Parameters

On systems based on the IA-32 architecture, the rounding flags control the method of rounding that the FPU uses. For example:

```
USE IFPORT

INTEGER(2) status, control, controlo, mask_all_traps

CALL GETCONTROLFPQQ(control)

WRITE (*, 9000) 'Control word: ', control

>

! Save old control word

controlo = control

! Clear the rounding control flags

control = IAND(control,NOT(FPCW$MCW_RC))

! Set new control to round up

control = IOR(control,FPCW$UP)

CALL SETCONTROLFPQQ(control)

CALL GETCONTROLFPQQ(control)

WRITE (*, 9000) 'Control word: ', control
```

The rounding options are mutually exclusive. If you set more than one, you may get an invalid mode or a mode other than the one you want. Therefore, you should clear the rounding bits before setting a new rounding mode.

NOTE. The GETCONTROLFPQQ and SETCONTROLFPQQ routines only affect the x87 status register. They do not affect the MXCSR register (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions).

Handling Floating-point Exceptions with the -fpe or /fpe Compiler Option

Using the -fpe or /fpe Compiler Options

The -fpe*n* (Linux* and Mac OS* X) or /fpe:*n* (Windows*) option allows some control over the results of floating-point exceptions.

-fpe0 or /fpe:0 restricts floating-point exceptions by enabling the overflow, the divide-by-zero, and the invalid floating-point exceptions. The program will print an error message and abort if any of these exceptions occurs. If a floating underflow occurs, the result is set to zero and execution continues. This is called flush-to-zero. This option sets -fp-speculation=strict (Linux and Mac OS X) or /Qfp-speculation:strict (Windows) if no specific -fp-speculation or /Qfp-speculation option is specified. The -fpe0 or /fpe:0 option sets -ftz (Linux and Mac OS X) /Qftz(Windows). To get more detailed location information about where the exception occurred, use -traceback (Linux and Mac OS X) or /traceback (Windows).

NOTE. On systems based on the IA-32 and Intel® 64 architectures , explicitly setting -fpe0 or /fpe:0 can degrade performance since the generated code stream must be synchronized after each floating-point instruction to allow for abrupt underflow fix-up.

-fpe1 or /fpe:1 restricts only floating-point underflow. Floating-point overflow, floating-point divide-by-zero, and floating-point invalid produce exceptional values (NaN and signed Infinities) and execution continues. If a floating-point underflow occurs, the result is set to zero and execution continues. The /fpe:1 option sets -ftz or /Qftz.

NOTE. On systems based on the IA-32 and Intel® 64 architectures , explicitly setting -fpe1 or /fpe:1 can degrade performance since the generated code stream must be synchronized after each floating-point instruction to allow for abrupt underflow fix-up.

-fpe3 or /fpe:3 is the default on all processors, which allows full floating-point exception behavior. Floating-point overflow, floating-point divide-by-zero, and floating-point invalid produce exceptional values (NaN and signed Infinities) and execution continues. Floating underflow is gradual: denormalized values are produced until the result becomes 0.

The `-fpe` or `/fpe` option enables exceptions in the Fortran main program only. The floating-point exception behavior set by the Fortran main program remains in effect throughout the execution of the entire program unless changed by the programmer. If the main program is not Fortran, the user can use the Fortran intrinsic `FOR_SET_FPE` to set the floating-point exception behavior.

When compiling different routines in a program separately, you should use the same value of *n* in `-fpe`*n* or `/fpe:`*n*.

An example follows:

```
       IMPLICIT NONE
       real*4 res_uflow, res_oflow
       real*4 res_dbyz, res_inv
       real*4 small, big, zero, scale
       small = 1.0e-30
       big = 1.0e30
       zero = 0.0
       scale = 1.0e-10
!      IEEE underflow condition (Underflow Raised)
       res_uflow = small * scale
       write(6,100)"Underflow: ",small, " *", scale, " = ", res_uflow
!      IEEE overflow condition (Overflow Raised)
       res_oflow = big * big
       write(6,100)"Overflow:", big, " *", big, " = ", res_oflow
!      IEEE divide-by-zero condition (Divide by Zero Raised)
       res_dbyz = -big / zero
       write(6,100)"Div-by-zero:", -big, " /", zero, " = ", res_dbyz
!      IEEE invalid condition (Invalid Raised)
       res_inv = zero / zero
       write(6,100)"Invalid:", zero, " /", zero, " = ", res_inv
 100   format(A14,E8.1,A2,E8.1,A2,E10.1)
       end
```

Consider the following command line:

`ifort fpe.f90 -fpe0 -fp-model strict -g` (Linux and Mac OS X)

`ifort fpe.f90 /fpe:0 /fp:strict /traceback` (Windows)

Output similar to the following should result:

Windows:

```
   Underflow: 0.1E-29 * 0.1E-09 =   0.0E+00
forrtl: error (72): floating overflow
Image              PC       Routine Line       Source
fpe.exe            0040115B Unknown Unknown Unknown
fpe.exe            0044DFC0 Unknown Unknown Unknown
fpe.exe            00433277 Unknown Unknown Unknown
kernel32.dll       7C816D4F Unknown Unknown Unknown
```

Linux and Mac OS X:

```
./a.out
   Underflow: 0.1E-29* 0.1E-09 =   0.0E+00
forrtl: error (72): floating overflow
Image      PC       Routine    Line       Source
a.out      0804A063 Unknown    Unknown Unknown
a.out      08049E78 Unknown    Unknown Unknown
Unknown    B746B748 Unknown    Unknown Unknown
a.out      08049D31 Unknown    Unknown Unknown
Aborted
```

The following command line uses `/fpe1`:

`ifort fpe.f90 -fpe1 -g` (Linux and Mac OS X)
`ifort fpe.f90 /fpe:1 /traceback` (Windows)

The following output is produced:

```
   Underflow: 0.1E-29 * 0.1E-09 =   0.0E+00
    Overflow: 0.1E+31 * 0.1E+31 = Infinity
 Div-by-zero: -0.1E+31 / 0.0E+00 = -Infinity
     Invalid: 0.0E+00 / 0.0E+00 = NaN
```

The following command line uses `/fpe3`:

`ifort fpe.f90 -fpe3 -g` (Linux and Mac OS X)

`ifort fpe.f90 /fpe:3 /traceback` (Windows)

The following output is produced:

```
Underflow: 0.1E-29 * 0.1E-09 =   0.1E-39

    Overflow: 0.1E+31 * 0.1E+31 = Infinity

 Div-by-zero: -0.1E+31 / 0.0E+00 = -Infinity

     Invalid: 0.0E+00 / 0.0E+00 = NaN
```

Understanding the Impact of Application Types

The full consequences of the `/fpe` option depend on the type of application you are building. You only get the full support for the chosen option setting in a Fortran Console or QuickWin/Standard Graphics application, assuming you do not override the default run-time exception handler. The work to achieve the full behavior is done partly by each of the default run-time handler, the Fortran compiler, the math library, the underlying hardware, and the operating system.

Floating-point Exceptions in Fortran Console, Fortran QuickWin, and Fortran Standard Graphics Applications

When you build a console application, the compiler generates a few calls at the beginning of your Fortran main program to Fortran run-time routines that initialize the environment, either with default options or in accordance with your selected compile-time options.

For floating-point exception handling, `/fpe:3` is the default. The run-time system initializes the hardware to mask all exceptions. With `/fpe:3`:

- The Intel Fortran run-time system does this initialization automatically (no call from the compiled code).
- The IA-32 architecture hardware automatically generates the default IEEE result for exceptional conditions.
- Because traps are masked with `/fpe:3`, there are no traps and you may see exceptional values like Nan's, Infinities, and denormalized numbers in your computations.

Users can poll the floating-point status word with `GETSTATUSFPQQ` to see if an exception has occurred and can clear the status register with `CLEARSTATUSFPQQ`.

If you specify /fpe:0, the compiler generates a call to an Intel Fortran run-time routine FOR_SET_FPE, with an argument that unmasks all floating-point traps in the floating-point control word. In this case, the hardware does not supply the default IEEE result. It traps to the operating system, which then looks for a condition handler.

In a Fortran console or Fortran QuickWin application, the Intel Fortran run-time system provides a default condition handler unless you establish your own. For all exceptions except underflow, the run-time system just prints out an error message and aborts the application. For underflow, the run-time system replaces the result with zero. This treatment of underflow with /fpe:0 is called *abrupt underflow* to 0 (zero), as opposed to *gradual underflow* to 0 provided with /fpe:3.

Fixing up underflow results to zero can significantly degrade the performance of your application based on IA-32 architecture. If you are experiencing a large number of underflows, consider changing your code to avoid underflows or consider masking underflow traps and allowing the hardware to operate on denormalized numbers. The IA-32 architecture based hardware is designed to operate correctly in the denormalized range and doing so is much faster than trapping to fix up a result to zero.

Another important point to understand about selecting the /fpe option is that the generated code must support the trapping mode. When an IA-32 architecture based floating-point instruction generates an exceptional result, you do not necessarily get a trap. The instruction must be followed by an fwait instruction or another floating-point operate instruction to cause the trap to occur.

The Intel Fortran compiler generates machine code to support these requirements in accordance with your selected /fpe option setting. In other words, the generated code must support the trapping mode. You can see this by compiling a simple test program and look at the machine code listing with /fpe:0 first, then /fpe:3. There are no fwait instructions in the /fpe:3 code. Even if you replace the default run-time exception handler with your own handler, you may still want to compile with /fpe:0 to generate code that supports trapping.

Floating-Point Exceptions in Fortran DLL Applications

In a DLL, there is no main Fortran program (unless you have written your main program in Fortran), so there is no automatic calling of run-time routines to initialize the environment. Even if you select /fpe:0, there is nothing that causes the run-time system to unmask traps in the hardware so you won't see traps. You will continue to see the hardware generated default IEEE results (Nan's, Infinities, and and denormalized numbers in your computations). The generated code will still do its part by supplying the fwait instructions, and so on, but unless the traps are unmasked somehow, no traps will occur. You can use SETCONTROLFPQQ or FOR_SET_FPE to unmask traps in the floating-point control word.

There is also no default exception handling in a DLL. The main application that calls the DLL must provide this, or the code in the DLL must provide something when it is called. Since underflow processing (fixup to 0, and so on) is done by the default Fortran run-time system handler, a DLL won't have that feature automatically.

A typical strategy is to compile with `/fpe:0`, but only unmask floating divide-by-zero, floating overflow, and floating invalid traps in the floating-point control word. By leaving floating-point underflow traps masked, the hardware will continue to provide gradual underflow, but other floating-point exceptions will generate traps, which the user then handles as desired.

Floating-Point Exceptions in Fortran Windows Applications

In a Fortran Windows application, the situation is similar to a Fortran DLL application. You define a WinMain routine in your Fortran code as the main entry point for your application. The Fortran run-time system does not define the main routine as it does in a Fortran console or Fortran QuickWin (or Standard Graphics) application. Your code is not protected by the default handler and the default run-time initialization routines are not called.

Understanding IEEE Floating-point Operations 40

Overview: Understanding IEEE Floating-point Standard

This version of Intel® Compiler uses a close approximation to the IEEE floating-point standard (ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985) unless otherwise stated. This standard is common to many microcomputer-based systems due to the availability of fast processors that implement the required characteristics.

This section outlines the characteristics of the standard and its implementation for the Intel Compilers. Except as noted, the description includes both the IEEE standard and the Intel Compiler implementation.

Floating-point Formats

The IEEE Standard 754 specifies values and requirements for floating-point representation (such as base 2). The standard outlines requirements for two formats: basic and extended, and for two word-lengths within each format: single and double.

Intel Fortran supports single-precision format (REAL(4)) and double-precision format (REAL(8)) floating-point numbers. At some levels of optimization, some floating-point numbers are stored in register file format (which equals 1 bit sign, 15 bit exponent, 64 bits of significand rounded to 53 bits), rather than being stored in IEEE single or double precision format. This can affect the values produced by some computations. The compiler option `-fp-model` (Linux* and Mac OS* X) or `/fp` (Windows*) can control how floating-point expressions are evaluated, thus leading to more predictable results.

Limitations of Numeric Conversion

The Intel® Fortran floating-point conversion solution is not expected to fulfill all floating-point conversion needs.

For instance, data fields in record structure variables (specified in a STRUCTURE statement) and data components of derived types (TYPE statement) are not converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified. With `EQUIVALENCE` statements, the data type of the variable named in the I/O statement is used.

If a program reads an I/O record containing multiple format floating-point fields into a single variable (such as an array) instead of their respective variables, the fields will not be converted. When they are later examined as separate fields by the program, they will remain in the binary format they were stored in on disk, unless the program is modified.

Conversions of the following file structure types are not supported:

- Binary data (`FORM='BINARY'`)
- Formatted data (`FORM='FORMATTED'`)
- Unformatted data (`FORM='UNFORMATTED'`) written by Microsoft* Fortran PowerStation or by Intel Fortran with the `/fpscomp:ioformat` compiler option in effect.

Special Values

This topic lists the special values of IEEE floating-point REALs that the Intel® Compiler supports and provides a brief description of each of them.

Signed Zero

Intel Fortran treats zero as signed by default. The sign of zero is the same as the sign of a nonzero number. If you use the intrinsic function `SIGN` with zero as the second argument, the sign of the zero will be transferred. Comparisons, however, consider +0 to be equal to -0.

A signed zero is useful in certain numerical analysis algorithms but in most applications the sign of zero is invisible. -0.0 is accepted as formatted input but is printed as formatted output only when the `-assume minus0` compiler option is used.

Denormalized Numbers

Denormalized numbers (denormals) fill the gap between the smallest positive normalized number and the smallest negative number. Otherwise only (+/-) 0 occurs in that interval. Denormalized numbers extend the range of computable results by allowing for gradual underflow.

Systems based on the IA-32 architecture support the `Denormal Operand` status flag, which when set, means that at least one of the input operands to a floating-point operation is a denormal. The `Underflow` status flag is set when a number loses precision and becomes a denormal.

Denormalized values can be read and printed with formatted I/O.

Signed Infinity

Infinities are the result of arithmetic in the limiting case of operands with arbitrarily large magnitude. They provide a way to continue when an overflow occurs. The sign of an infinity is simply the sign you obtain for a finite number in the same operation as the finite number approaches an infinite value.

By retrieving the status flags, you can differentiate between an infinity that results from an overflow and one that results from division by zero. Intel® Compiler treats infinity as signed by default. +Infinity and -Infinity are accepted as formatted input and those strings are printed on/as? formatted output.

Not a Number

Not a Number (NaN) results from an invalid operation. For instance 0/0 and SQRT(-1) result in NaN. In general, an operation involving a NaN produces another NaN. Because the fraction of a NaN is unspecified, there are many possible NaNs. The Intel® processor treats all NaNs identically but provides two different types of NaNs:

- **Signaling NAN:** has an initial fraction bit of 0 (zero), which usually raises an invalid exception when used in an operation.
- **Quiet NaN:** has an initial fraction bit of 1.

The floating-point hardware changes a signaling NAN into a quiet NAN during many arithmetic operations, including the assignment operation. An invalid exception may be raised but the resulting floating-point value will be a quiet NAN. An operation may raise an invalid exception and return a signaling NaN but if the exception is not trapped, the signaling NaN may be turned into a quite NaN by subsequent processing.

Fortran binary and unformatted input and output do not change the internal representations of the values as they are handled. Therefore, signaling and quiet NANs may be read into real data and output to files in binary form. 'NaN' is accepted as formatted input and results in a quiet NaN. 'NaN' is printed by formatted output for both signaling and quiet NaNs.

Representing Floating-point Numbers

Floating-point Representation

The Fortran numeric environment is flexible, which helps make Fortran a strong language for intensive numerical calculations. The Fortran standard purposely leaves the precision of numeric quantities and the method of rounding numeric results unspecified. This allows Fortran to operate efficiently for diverse applications on diverse systems.

Computations on real numbers may not yield what you expect. This happens because the hardware must represent numbers in a finite number of bits.

There are several effects of using finite floating-point numbers. The hardware is not able to represent every real number exactly, but must approximate exact representations by rounding or truncating to finite length. In addition, some numbers lie outside the range of representation of the maximum and minimum exponents and can result in calculations that underflow and overflow. As an example of one consequence, finite precision produces many numbers that, although non-zero, behave in addition as zero.

You can minimize the effects of finite representation with programming techniques; for example, by not using floating-point numbers in LOGICAL comparisons or by giving them a tolerance (for example, IF (ABS(x-10.0) <= 0.001)), and by not attempting to combine or compare numbers that differ by more than the number of significant bits.

Floating-point numbers approximate real numbers with a finite number of bits. The bits are calculated as shown in the following formula. The representation is binary, so the base is 2. The bits b_n represent binary digits (0 or 1). The precision *P* is the number of bits in the nonexponential part of the number (the significand), and *E* is the exponent. With these parameters, binary floating-point numbers approximate real numbers with the values:

$(- 1)^s b_0 . b_1 b_2 \ldots b_{P-1} \times 2^E$

where *s* is 0 or 1 (+ or -), and $E_{min} <= E <= E_{max}$

The following table gives the standard values for these parameters for single, double, and quad (extended precision) formats and the resulting bit widths for the sign, the exponent, and the full number.

Parameters for IEEE* Floating-Point Formats

Parameter	Single	Double	Quad or Extended Precision (IEEE_X)*
Sign width in bits	1	1	1

Parameter	Single	Double	Quad or Extended Precision (IEEE_X)*
P	24	53	113
E_{max}	+127	+1023	+16383
E_{min}	-126	-1022	-16382
Exponent*bias*	+127	+1023	+16383
Exponent width in bits	8	11	15
Format width in bits	32	64	128

* This type is emulated in software.

The actual number of bits needed to represent the precisions 24, 53, and 113 is therefore 23, 52, and 112, respectively, because b_0 is chosen to be 1 implicitly.

A *bias* is added to all exponents so that only positive integer exponents occur. This expedites comparisons of exponent values. The stored exponent is actually:

$e = E + bias$

See Also

- Representing Floating-point Numbers
- Native IEEE Floating-Point Representations
- Rounding Errors

Retrieving Parameters of Numeric Representations

Intel Fortran includes several intrinsic functions that return details about the numeric representation. These are listed in the following table and described fully in the *Language Reference*.

Functions that Return Numeric Parameters

Name	Description	Argument/Function Type
DIGITS	DIGITS(*x*). Returns number of significant digits for data of the same type as *x*.	*x*: Integer or Real result: INTEGER(4)
EPSILON	EPSILON(*x*). Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as *x*.	*x*: Real result: same type as *x*
EXPONENT	EXPONENT(*x*). Returns the exponent part of the representation of *x*.	*x*: Real result: INTEGER(4)
FRACTION	FRACTION(*x*). Returns the fractional part (significand) of the representation of *x*.	*x*: Real result: same type as *x*
HUGE	HUGE(*x*). Returns largest number that can be represented by data of type *x*.	*x*: Integer or Real result: same type as *x*.
MAXEXPONENT	MAXEXPONENT(*x*). Returns the largest positive decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
MINEXPONENT	MINEXPONENT(*x*). Returns the largest negative decimal exponent for data of the same type as *x*.	*x*: Real result: INTEGER(4)
NEAREST	NEAREST(*x, s*). Returns the nearest different machine representable number to *x* in the direction of the sign of *s*.	*x*: Real *s*: Real and not zero result: same type as *x*.
PRECISION	PRECISION(*x*). Returns the number of significant digits for data of the same type as *x*.	*x*: Real or Complex result: INTEGER(4)
RADIX	RADIX(*x*). Returns the base for data of the same type as *x*.	*x*: Integer or Real result: INTEGER(4)

Name	Description	Argument/Function Type
RANGE	RANGE(*x*). Returns the decimal exponent range for data of the same type as *x*.	*x*: Integer, Real or Complex result: INTEGER(4)
RRSPACING	RRSPACING(*x*). Returns the reciprocal of the relative spacing of numbers near *x*.	*x*: Real result: same type as *x*
SCALE	SCALE(*x, i*). Multiplies *x* by 2 raised to the power of *i*.	*x*: Real *i*: Integer result: same type as *x*
SET_EXPONENT	SET_EXPONENT(*x,i*). Returns a number whose fractional part is *x* and whose exponential part is *i*.	*x*: Real *i*: Integer result: same type as *x*
SPACING	SPACING(*x*). Returns the absolute spacing of numbers near *x*.	*x*: Real result: same type as *x*
TINY	TINY(*x*). Returns smallest positive number that can be represented by data of type *x*.	*x*: Real result: same type as *x*

ULPs, Relative Error, and Machine Epsilon

ULP, Relative Error, and Machine Epsilon are terms that describe the magnitude of rounding error. A floating-point approximation to a real constant or to a computed result may err by as much as 1/2 unit in the last place (the b_{P-1} bit). The abbreviation *ULP* represents the measure "unit in the last place." Another measure of the rounding error uses the relative error, which is the difference between the exact number and its approximation divided by the exact number. The relative error that corresponds to 1/2 *ULP* is bounded by:

$1/2\ 2^{-P} <= 1/2\ ULP <= 2^{-P}$

The upper bound $EPS = 2^{-P}$, the machine epsilon, is commonly used in discussions of rounding errors because it expresses the smallest floating-point number that you can add to 1.0 with a result that does not round to 1.0.

See Also

- Representing Floating-point Numbers
- Floating-point Representation for 'b' and 'P' definitions

Native IEEE Floating-point Representation

Overview: Native IEEE* Floating-point Representations

The REAL(4) (IEEE* S_floating), REAL(8) (IEEE T_floating), and REAL(16) (IEEE-style X_floating) formats are stored in standard little endian IEEE binary floating-point notation. (See IEEE Standard 754 for additional information about IEEE binary floating point notation.) COMPLEX() formats use a pair of REAL values to denote the real and imaginary parts of the data.

All floating-point formats represent fractions in sign-magnitude notation, with the binary radix point to the right of the most-significant bit. Fractions are assumed to be normalized, and therefore the most-significant bit is not stored (this is called "hidden bit normalization"). This bit is assumed to be 1 unless the exponent is 0. If the exponent equals 0, then the value represented is denormalized (subnormal) or plus or minus zero.

Intrinsic REAL kinds are 4 (single precision), 8 (double precision), and 16 (extended precision), such as REAL(KIND=4) for single-precision floating-point data. Intrinsic COMPLEX kinds are also 4 (single precision), 8 (double precision), and 16 (extended precision).

To obtain the kind of a variable, use the KIND intrinsic function. You can also use a size specifier, such as REAL*4, but be aware this is an extension to the Fortran 2003 standard.

If you omit certain compiler options, the default sizes for REAL and COMPLEX data declarations are as follows:

- For REAL data declarations without a kind parameter (or size specifier), the default size is REAL (KIND=4) (same as REAL*4).
- For COMPLEX data declarations without a kind parameter (or size specifier), the default data size is COMPLEX (KIND=4) (same as COMPLEX*8).

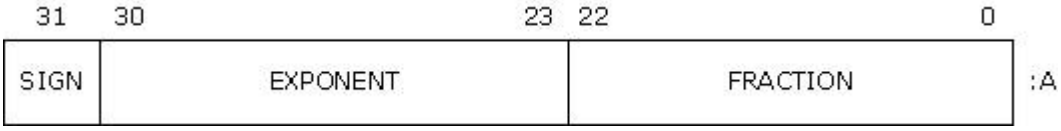
To control the size of all REAL or COMPLEX declarations without a kind parameter, use the `/real_size:64` (or `/4R8`) or /real_size:128 (or `/4R16`) options; the default is `/real_size:32`.

You can explicitly declare the length of a REAL or a COMPLEX declaration using a kind parameter, or specify DOUBLE PRECISION or DOUBLE COMPLEX. To control the size of all DOUBLE PRECISION and DOUBLE COMPLEX declarations, use the `/double_size:128` (or `/Qautodouble`) option; the default is `/double_size:64`.

REAL(KIND=4) (REAL) Representation

REAL(4) (same as REAL(KIND=4)) data occupies 4 contiguous bytes stored in IEEE S_floating format. Bits are labeled from the right, 0 through 31, as shown below.

REAL(4) Floating-Point Data Representation



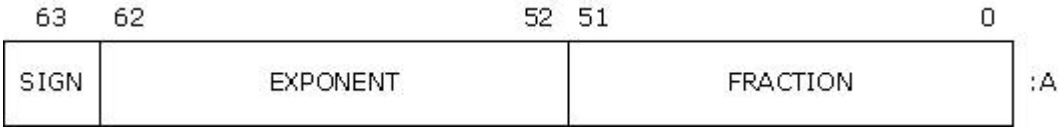
The form of REAL(4) data is sign magnitude, with bit 31 the sign bit (0 for positive numbers, 1 for negative numbers), bits 30:23 a binary exponent in excess 127 notation, and bits 22:0 a normalized 24-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 1.17549435E-38 (normalized) to 3.40282347E38. The IEEE denormalized (subnormal) limit is 1.40129846E-45. The precision is approximately one part in 2**23; typically 7 decimal digits.

REAL(KIND=8) (DOUBLE PRECISION) Representation

REAL(8) (same as REAL(KIND=8)) data occupies 8 contiguous bytes stored in IEEE T_floating format. Bits are labeled from the right, 0 through 63, as shown below.

REAL(8) Floating-Point Data Representation

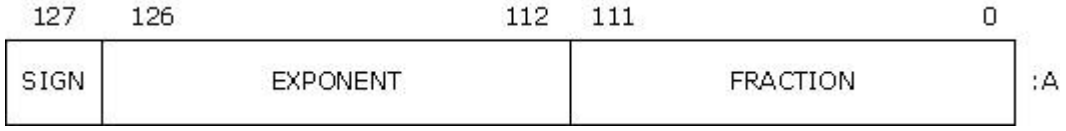


The form of REAL(8) data is sign magnitude, with bit 63 the sign bit (0 for positive numbers, 1 for negative numbers), bits 62:52 a binary exponent in excess 1023 notation, and bits 51:0 a normalized 53-bit fraction including the redundant most-significant fraction bit not represented.

The value of data is in the approximate range: 2.2250738585072013D-308 (normalized) to 1.7976931348623158D308. The IEEE denormalized (subnormal) limit is 4.94065645841246544D-324. The precision is approximately one part in 2**52; typically 15 decimal digits.

REAL(KIND=16) Representation

REAL(16) (same as REAL(KIND=16)) data occupies 16 contiguous bytes stored in IEEE-style X_floating format. Bits are labeled from the right, 0 through 127, as shown below.



The form of REAL(16) data is sign magnitude, with bit 127 the sign bit (0 for positive numbers, 1 for negative numbers), bits 126:112 a binary exponent in excess 16383 notation, and bits 111:0 a normalized 113-bit fraction including the redundant most-significant fraction bit not represented.

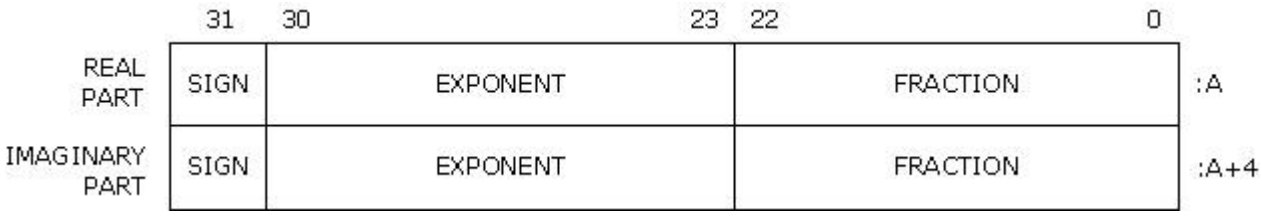
The value of data is in the approximate range: 6.475175119438025110924438958227646552499 6Q-4966 to 1.189731495357231765085759326628007016196477Q4932. Unlike other floating-point formats, there is little if any performance penalty from using denormalized extended-precision numbers. This is because accessing denormalized REAL (KIND=16) numbers does not result in an arithmetic trap (the extended-precision format is emulated in software). The smallest normalized number is 3.362103143112093506262677817321753Q-4932.

The precision is approximately one part in 2**112 or typically 33 decimal digits.

COMPLEX(KIND=4) (COMPLEX) Representation

COMPLEX(4) (same as COMPLEX(KIND=4) and COMPLEX*8) data is 8 contiguous bytes containing a pair of REAL(4) values stored in IEEE S_floating format. The low-order 4 bytes contain REAL(4) data that represents the real part of the complex number. The high-order 4 bytes contain REAL(4) data that represents the imaginary part of the complex number, as shown below.

COMPLEX(4) Floating-Point Data Representation

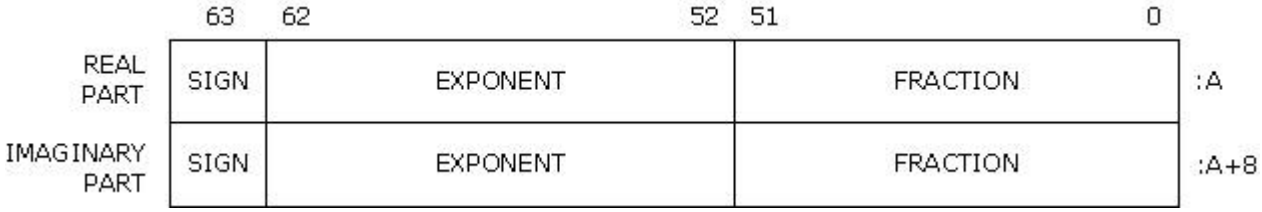


The limits and underflow characteristics for `REAL(4)` apply to the two separate real and imaginary parts of a COMPLEX(4) number. Like REAL(4) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

COMPLEX(KIND=8) (DOUBLE COMPLEX) Representation

COMPLEX(8) (same as COMPLEX(KIND=8) and COMPLEX*16) data is 16 contiguous bytes containing a pair of REAL(8) values stored in IEEE T_floating format. The low-order 8 bytes contain REAL(8) data that represents the real part of the complex data. The high-order 8 bytes contain REAL(8) data that represents the imaginary part of the complex data, as shown below.

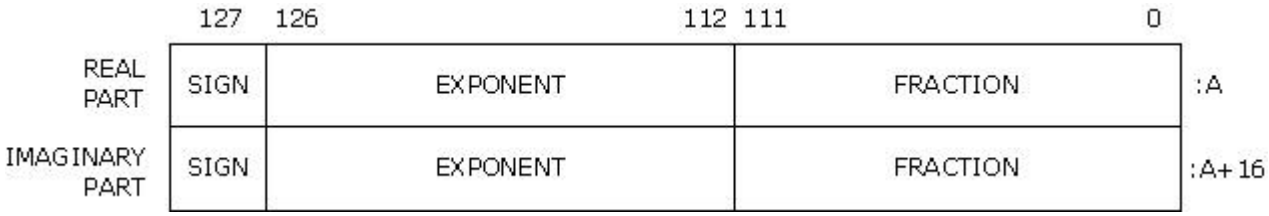
COMPLEX(8) Floating-Point Data Representation



The limits and underflow characteristics for REAL(8) apply to the two separate real and imaginary parts of a COMPLEX(8) number. Like REAL(8) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

COMPLEX(KIND=16) Representation

COMPLEX(16) (same as COMPLEX(KIND=16) or COMPLEX*32) data is 32 contiguous bytes containing a pair of REAL(16) values stored in IEEE-style X_floating format. The low-order 16 bytes contain REAL(16) data that represents the real part of the complex data. The high-order 16 bytes contain REAL(8) data that represents the imaginary part of the complex data, as shown below.



The limits and underflow characteristics for REAL(16) apply to the two separate real and imaginary parts of a COMPLEX(16) number. Like REAL(16) numbers, the sign bit representation is 0 (zero) for positive numbers and 1 for negative numbers.

Handling Exceptions and Errors

Loss of Precision Errors

If a real number is not exactly one of the representable floating-point numbers, then the nearest floating-point number must represent it. The rounding error is the difference between the exact real number and its nearest floating-point representation. If the rounding error is non-zero, the rounded floating-point number is called *inexact*.

Normally, calculations proceed when an inexact value results. Almost any floating-point operation can produce an inexact result. The rounding mode (round up, round down, round nearest, truncate) is determined by the floating-point control word.

If an arithmetic operation results in a floating-point number that cannot be represented in a specific data type, the operation may produce a special value: signed zero, signed infinity, NaN, or a denormal. Numbers that have been rounded to an exactly representable floating-point number also result in a special value. Special-value results are a limiting case of the arithmetic operation involved. Special values can propagate through your arithmetic operations without causing your program to fail, and often provide usable results.

If an arithmetic operation results in an exception, the operation can cause an underflow or overflow:

- Underflow occurs when an arithmetic result is too small for the math processor to handle. Depending on the setting of the `/fpe` compiler option, underflows are set to zero (they are usually harmless) or they are left as is (denormalized).
- Overflow occurs when an arithmetic result is too large for the math processor to handle. Overflows are more serious than underflows, and may indicate an error in the formulation of a problem (for example, unintended exponentiation of a large number by a large number). Overflows generally produce an appropriately signed infinity value. (This depends on the rounding mode as per the IEEE standard.)

An arithmetic operation can also throw the following exceptions: divide-by-zero exception, an invalid exception, and an inexact exception.

You can select how exceptions are handled by setting the floating-point control word.

See Also

- Handling Exceptions and Errors
- Special Values

Rounding Errors

Although the rounding error for one real number might be acceptably small in your calculations, at least two problems can arise because of it. If you test for exact equality between what you consider to be two exact numbers, the rounding error of either or both floating-point representations of those numbers may prevent a successful comparison and produce spurious results. Also, when you calculate with floating-point numbers the rounding errors may accumulate to a meaningful loss of numerical significance.

Carefully consider the numerics of your solution to minimize rounding errors or their effects. You might benefit from using double-precision arithmetic or restructuring your algorithm, or both. For instance, if your calculations involve arrays of linear data items, you might reduce the loss of numerical significance by subtracting the mean value of each array from each array element and by normalizing each element of such an array to the standard deviation of the array elements.

The following code segment can execute differently on various systems and produce varying results for `n`, `x`, and `s`. It also produces different results if you use the `-fp-model precise` (Linux* and Mac OS* X) or `/fp:precise` (Windows*; systems), or `-fp-model fast` (Linux and Mac OS X) or `/fp:fast` (Windows) compiler options. Rounding error accumulates in `x` because the floating-point representation of 0.2 is inexact, then accumulates in `s`, and affects the final value for `n`:

```
  INTEGER n
  REAL s, x
  n = 0
  s = 0.0
  x = 0.0
1 n = n + 1
  x = x + 0.2
  s = s + x
  IF ( x .LE. 10. ) GOTO 1 ! Will you get 51 cycles?
  WRITE(*,*) 'n = ', n, '; x = ', x, '; s = ', s
```

This example illustrates a common coding problem: carrying a floating-point variable through many successive cycles and then using it to perform an `IF` test. This process is common in numerical integration. There are several remedies. You can compute `x` and `s` as multiples of an integer index, for example, replacing the statement that increments `x` with `x = n * 0.2`

to avoid round-off accumulation. You might test for completion on the integer index, such as `IF (n <= 50) GOTO 1`, or use a `DO` loop, such as `DO n= 1,51`. If you must test on the real variable that is being cycled, use a realistic tolerance, such as `IF (x <= 10.001)`.

Floating-point arithmetic does not always obey the standard rules of algebra exactly. Addition is not precisely associative when round-off errors are considered. You can use parentheses to express the exact evaluation you require to compute a correct, accurate answer, provided you use a switch such as `/fp:precise` (on Windows systems) or `-fp-model precise` (on Linux or MacOS X systems), or `/assume:protect_parens` (on Windows systems) or `-assume protect_parens` (on Linux or MacOS X systems). This is recommended when you specify optimization for your generated code, since associativity may otherwise be unpredictable.

The expressions (x + y) + z and x + (y + z) can give unequal results in some cases.

The compiler uses the default rounding mode (round-to-nearest) during compilation. The compiler performs more compile-time operations that eliminate runtime operations as the optimization level increases. If you set rounding mode to a different setting (other than round-to-nearest), that rounding mode is used only if that computation is performed at runtime. If you want to force computations to be performed at runtime, use the `-fp-model strict` (Linux or MacOS) or `/fp:strict` (Windows) option.

See Also

- Handling Exceptions and Errors
- ULPs, Relative Error, and Machine Epsilon

Part V

Language Reference

Topics:

- Overview: Language Reference
- Conformance, Compatibility, and Fortran 2003 Features
- Program Structure, Characters, and Source Forms
- Data Types, Constants, and Variables
- Expressions and Assignment Statements
- Specification Statements
- Dynamic Allocation
- Execution Control
- Program Units and Procedures
- Intrinsic Procedures
- Data Transfer I/O Statements
- I/O Formatting
- File Operation I/O Statements
- Compilation Control Lines and Statements
- Directive Enhanced Compilation
- Scope and Association
- Deleted and Obsolescent Language Features
- Additional Language Features
- Additional Character Sets
- Data Representation Models
- Run-Time Library Routines

- Summary of Language Extensions
- A to Z Reference
- Glossary

Overview: Language Reference 41

This document contains the complete description of the Intel® Fortran programming language, which includes Fortran 95, Fortran 90, and many Fortran 2003 language features. It contains information on language syntax and semantics, on adherence to various Fortran standards, and on extensions to those standards.

This manual is intended for experienced applications programmers who have a basic understanding of Fortran concepts and the Fortran 95/90 language.

Some familiarity with your operating system is helpful. This manual is not a Fortran or programming tutorial.

This manual contains the full content of what originally appeared in the Language Reference and Libraries Reference PDF files. However, the library routines described in the Libraries Reference PDF file are now described within the A to Z Reference.

This document covers the following topics:

- Conformance, Compatibility, and Fortran 2003 Features

  This topic describes conformance with language standards, compatibility with other Fortran languages, and it contains a summary of Fortran 2003 features.
- Program Structure, Characters, and Source Forms

  This topic describes program structure, the Fortran 95/90 character set, and source forms.
- Data Types, Constants, and Variables

  This topic describes language standards, language compatibility, and Fortran 95/90 features.
- Expressions and Assignment Statements

  This topic describes Fortran expressions and assignment statements, which are used to define or redefine variables.
- Specification Statements

  This topic describes specification statements, which are used to declare the attributes of data objects.
- Dynamic Allocation

  This topic describes statements used in dynamic allocation: ALLOCATE, DEALLOCATE, and NULLIFY.
- Execution Control

  This topic describes constructs and statements that can transfer control within a program.
- Program Units and Procedures

  This topic describes program units (including modules), subroutines and functions, and procedure interfaces.

- Intrinsic Procedures

  This topic describes argument keywords used in intrinsic procedures and provides an overview of intrinsic procedures.
- Data Transfer I/O Statements

  This topic describes data transfer input/output (I/O) statements.
- I/O Formatting

  This topic describes the rules for I/O formatting.
- File Operation I/O Statements

  This topic describes auxiliary I/O statements you can use to perform file operations.
- Compilation Control Statements

  This topic describes compilation control statements: INCLUDE and OPTIONS.
- Directive Enhanced Compilation

  This topic describes general and parallel compiler directives.
- Scope and Association

  This topic describes scope, which refers to the area in which a name is recognized, and association, which is the language concept that allows different names to refer to the same entity in a particular region of a program.
- Deleted and Obsolescent Language Features

  This topic describes deleted features in Fortran 95 and obsolescent language features in Fortran 95 and Fortran 90.
- Additional Language Features

  This topic describes some statements and language features supported for programs written in older versions of Fortran.
- Additional Character Sets

  This topic describes the additional character sets available on Windows*, Linux*, and Mac OS* X systems.
- Data Representation Models

  This topic describes data representation models for numeric intrinsic functions.
- Run-Time Library Routines

  This topic summarizes the many run-time library routines.
- Summary of Language Extensions

  This topic summarizes Intel Fortran extensions to the Fortran 95 Standard.
- A to Z Reference

This topic contains language summary tables and descriptions of all Intel® Fortran statements, intrinsics, directives, and module library routines, which are listed in alphabetical order.

- Glossary

  This topic contains abbreviated definitions of some commonly used terms in this manual.

For details on the features of the compilers, see your guide to *Building Applications*. For details on how to improve the run-time performance of Fortran programs, see your guide to *Optimizing Applications*. For details on floating-point support, see your guide to Floating-point Operations. For details on compiler options, see your *Compiler Options* reference.

For information on conventions used in this document, see Conventions.

For more information on Fortran 2003 features in this release, see Fortran 2003 Features.

New Language Features

The major new features for this release are as follows:

- OpenMP* Fortran directive TASK

  The TASK directive defines a task region. For more information, see TASK.

- OpenMP* Fortran directive TASKWAIT

  The TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task. For more information, see TASKWAIT.

- OpenMP* Fortran directive clause SCHEDULE (AUTO)

  The AUTO setting delegates the scheduling decision until compile time or run time. The schedule is processor dependent. For more information, see SCHEDULE in the DO directive.

- VECTOR TEMPORAL directive

  The VECTOR TEMPORAL directive tells the compiler to use temporal (that is, non-streaming) stores. For more information, see VECTOR TEMPORAL and VECTOR NONTEMPORAL.

- VECTOR NONTEMPORAL directive now allows variables

  VECTOR NONTEMPORAL directs the compiler to use non-temporal (that is, streaming) stores. It now allows variables as optional memory references. For more information, see VECTOR TEMPORAL and VECTOR NONTEMPORAL.

- UNROLL_AND_JAM and NOUNROLL_AND_JAM directives

  The UNROLL_AND_JAM and NOUNROLL_AND_JAM directives enable or disable loop unrolling and jamming. For more information, see UNROLL_AND_JAM and NOUNROLL_AND_JAM.

For more information on Fortran 2003 features, see Fortran 2003 Features.

For information on new compiler options in this release, see New Options in the *Compiler Options* reference.

Conformance, Compatibility, and Fortran 2003 Features

42

Fortran 95 includes Fortran 90 and most features of FORTRAN 77. Fortran 90 is a superset that includes FORTRAN 77. Intel Fortran fully supports the Fortran 95, Fortran 90, and FORTRAN 77 Standards.

Language Standards Conformance

Intel Fortran conforms to American National Standard Fortran 95 (ANSI X3J3/96-007)[1], American National Standard Fortran 90 (ANSI X3.198-1992)[2], and includes support for many features in the Fortran 2003 standard (ISO/IEC 1539-1:2004).

The ANSI committee X3J3 is currently answering questions of interpretation of Fortran 95 and Fortran 90 language features. Any answers given by the ANSI committee that are related to features implemented in Intel Fortran may result in changes in future releases of the Intel Fortran compiler, even if the changes produce incompatibilities with earlier releases of Intel Fortran.

Intel Fortran provides a number of extensions to the Fortran 95 Standard. In the language reference, extensions are displayed in this color.

Intel Fortran also includes support for programs that conform to the previous Fortran standards (ANSI X3.9-1978 and ANSI X3.0-1966), the International Standards Organization standard ISO 1539-1980 (E), the Federal Information Processing Institute standard FIPS 69-1, and the Military Standard 1753 Language Specification.

Language Compatibility

Intel Fortran is highly compatible with Compaq* Fortran 77 on supported systems, and it is substantially compatible with PDP-11* and VAX* FORTRAN 77.

Fortran 2003 Features

The following Fortran 2003 features are new in this release:

- Enumerators
- Type extension (not polymorphic)
- Allocatable scalar variables (not deferred-length character)
- ERRMSG keyword for ALLOCATE and DEALLOCATE

[1] This is the same as International Standards Organization standard ISO/IEC 1539-1:1997 (E).
[2] This is the same as International Standards Organization standard ISO/IEC 1539:1991 (E).

- SOURCE= keyword for ALLOCATE
- Character arguments for MAX, MIN, MAXVAL, MINVAL, MAXLOC, and MINLOC
- Intrinsic modules IEEE_EXCEPTIONS, IEEE_ARITHMETIC and IEEE_FEATURES
- ASSOCIATE construct
- PROCEDURE declaration
- Procedure pointers
- ABSTRACT INTERFACE
- PASS and NOPASS attributes
- Structure constructors with component names and default initialization
- Array constructors with type and character length specifications
- I/O keywords BLANK, DELIM, ENCODING, IOMSG, PAD, ROUND, SIGN, and SIZE
- Format edit descriptors DC, DP, RD, RC, RN, RP, RU, and RZ

The following Fortran 2003 features are also supported:

- RECORDTYPE setting STREAM_CRLF
- A file can be opened for stream access (ACCESS='STREAM')
- Specifier POS can be specified in an INQUIRE, READ, or WRITE statement
- BIND attribute and statement
- Language binding can be specified in a FUNCTION or SUBROUTINE statement, or when defining a derived type
- IS_IOSTAT_END intrinsic function
- IS_IOSTAT_EOR intrinsic function
- INTRINSIC and NONINTRINSIC can be specified for modules in USE statements
- ASYNCHRONOUS attribute and statement
- VALUE attribute and statement
- Specifier ASYNCHRONOUS can be specified in an OPEN, INQUIRE, READ, or WRITE statement
- An ID can be specified for a pending data transfer operation
- FLUSH statement
- WAIT statement
- IMPORT statement
- NEW_LINE intrinsic function
- SELECTED_CHAR_KIND intrinsic function

- Intrinsic modules ISO_C_BINDING and ISO_FORTRAN_ENV
- MEMORYTOUCH compiler directive
- Specifiers ID and PENDING can be specified in an INQUIRE statement
- User-defined operators can be renamed in USE statements
- MOVE_ALLOC intrinsic subroutine
- PROTECTED attribute and statement
- Pointer objects can have the INTENT attribute
- GET_COMMAND intrinsic
- GET_COMMAND_ARGUMENT intrinsic
- COMMAND_ARGUMENT_COUNT intrinsic
- GET_ENVIRONMENT_VARIABLE intrinsic
- Allocatable components of derived types
- Allocatable dummy arguments
- Allocatable function results
- VOLATILE attribute and statement
- Names of length up to 63 characters
- Statements up to 256 lines
- A named PARAMETER constant may be part of a complex constant
- In all I/O statements, the following numeric values can be of any kind: UNIT=, IOSTAT=
- The following OPEN numeric values can be of any kind: RECL=
- The following READ and WRITE numeric values can be of any kind: REC=, SIZE=
- The following INQUIRE numeric values can be of any kind: NEXTREC=, NUMBER=, RECL=, SIZE=
- Recursive I/O is allowed when the new I/O being started is internal I/O that does not modify any internal file other than its own
- IEEE infinities and Nans are displayed by formatted output as specified by Fortran 2003
- In an I/O format, the comma after a P edit descriptor is optional when followed by a repeat specifier
- The following intrinsics take an optional KIND= argument: ACHAR, COUNT, IACHAR, ICHAR, INDEX, LBOUND, LEN, LEN_TRIM, MAXLOC, MINLOC, SCAN, SHAPE, SIZE, UBOUND, VERIFY
- Square brackets [ ] are permitted to delimit array constructors instead of (/ /)
- The Fortran character set has been extended to contain the 8-bit ASCII characters ~ \ [ ] ` ^ { } | # @

See Also

- Conformance, Compatibility, and Fortran 2003 Features
- New Language Features

Program Structure, Characters, and Source Forms

43

This section contains information on the following topics:

- An overview of program structure, including general information on statements and names
- Character sets
- Source forms

Program Structure

A Fortran program consists of one or more program units. A *program unit* is usually a sequence of statements that define the data environment and the steps necessary to perform calculations; it is terminated by an END statement.

A program unit can be either a main program, an external subprogram, a module, or a block data program unit. An executable program contains one main program, and, optionally, any number of the other kinds of program units. Program units can be separately compiled.

An *external subprogram* is a function or subroutine that is not contained within a main program, a module, or another subprogram. It defines a procedure to be performed and can be invoked from other program units of the Fortran program. Modules and block data program units are not executable, so they are not considered to be procedures. (Modules can contain module procedures, though, which are executable.)

Modules contain definitions that can be made accessible to other program units: data and type definitions, definitions of procedures (called *module subprograms*), and *procedure interfaces*. Module subprograms can be either functions or subroutines. They can be invoked by other module subprograms in the module, or by other program units that access the module.

A *block data program unit* specifies initial values for data objects in named common blocks. In Fortran 95/90, this type of program unit can be replaced by a module program unit.

Main programs, external subprograms, and module subprograms can contain *internal subprograms*. The entity that contains the internal subprogram is its *host*. Internal subprograms can be invoked only by their host or by other internal subprograms in the same host. Internal subprograms must not contain internal subprograms.

The following sections discuss Statements, Names, and Keywords.

Statements

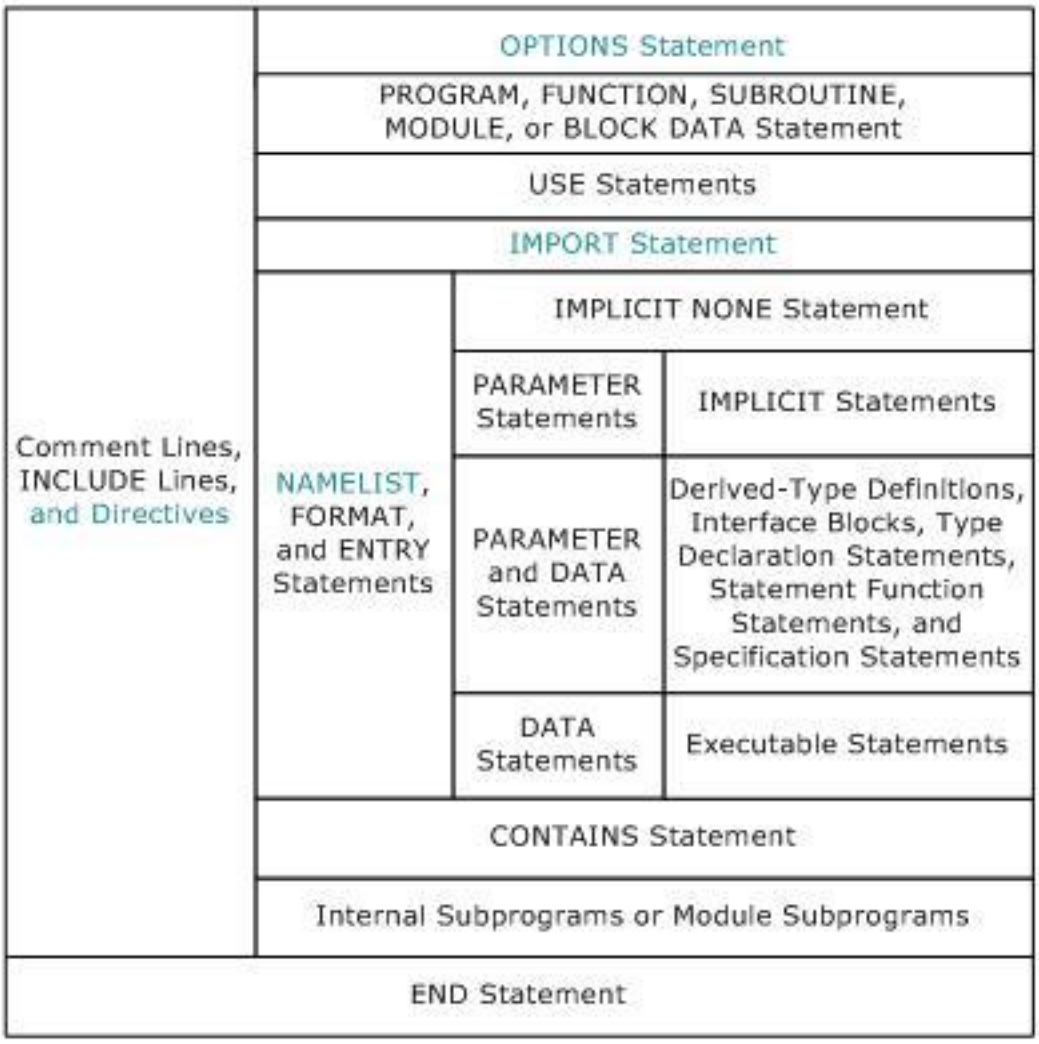
Program statements are grouped into two general classes: executable and nonexecutable. An *executable statement* specifies an action to be performed. A *nonexecutable statement* describes program attributes, such as the arrangement and characteristics of data, as well as editing and data-conversion information.

Order of Statements in a Program Unit

The following figure shows the required order of statements in a Fortran program unit. In this figure, vertical lines separate statement types that can be interspersed. For example, you can intersperse DATA statements with executable constructs.

Horizontal lines indicate statement types that cannot be interspersed. For example, you cannot intersperse DATA statements with CONTAINS statements.

Required Order of Statements



PUBLIC and PRIVATE statements are only allowed in the scoping units of modules. In Fortran 95/90, NAMELIST statements can appear only among specification statements. However, Intel® Fortran allows them to also appear among executable statements.

The following table shows other statements restricted from different types of scoping units.

Statements Restricted in Scoping Units

Scoping Unit	Restricted Statements
Main program	ENTRY, IMPORT, and RETURN statements
Module[1]	ENTRY, FORMAT, IMPORT, OPTIONAL, and INTENT statements, statement functions, and executable statements
Block data program unit	CONTAINS, ENTRY, IMPORT, and FORMAT statements, interface blocks, statement functions, and executable statements
Internal subprogram	CONTAINS, IMPORT, and ENTRY statements
Interface body	CONTAINS, DATA, ENTRY, IMPORT[2], SAVE, and FORMAT statements, statement functions, and executable statements

[1] The scoping unit of a module does not include any module subprograms that the module contains.

[2] An IMPORT statement can appear only in the *interface-body* of an INTERFACE block.

See Also

- Program Structure
- Scope

Names

Names identify entities within a Fortran program unit (such as variables, function results, common blocks, named constants, procedures, program units, namelist groups, and dummy arguments). In FORTRAN 77, names were called "symbolic names".

A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign.

In Fortran 95/90, a name can contain up to 31 characters. Intel® Fortran allows names up to 63 characters.

The length of a module name (in MODULE and USE statements) may be restricted by your file system.

NOTE. Be careful when defining names that contain dollar signs. A dollar sign can be a symbol for command or symbol substitution in various shell and utility commands.

In an executable program, the names of the following entities are global and must be unique in the entire program:

- Program units
- External procedures
- Common blocks
- Modules

Examples

The following examples demonstrate valid and invalid names

Table 520: Valid Names

NUMBER
FIND_IT
X

Table 521: Invalid Names

5Q	Begins with a numeral.
B.4	Contains a special character other than _ or $.
_WRONG	Begins with an underscore.

The following are all valid examples of using names:

```
INTEGER (SHORT) K       !K names an integer variable
SUBROUTINE EXAMPLE      !EXAMPLE names the subroutine
LABEL: DO I = 1,N       !LABEL names the DO block
```

Keywords

A keyword can either be a part of the syntax of a statement (statement keyword), or it can be the name of a dummy argument (argument keyword). Examples of statement keywords are WRITE, INTEGER, DO, and OPEN. Examples of argument keywords are arguments to the intrinsic functions.

In the intrinsic function UNPACK (*vector*, *mask*, *field*), for example, *vector*, *mask*, and *field* are argument keywords. They are dummy argument names, and any variable may be substituted in their place. Dummy argument names and real argument names are discussed in Program Units and Procedures.

Keywords are not reserved. The compiler recognizes keywords by their context. For example, a program can have an array named IF, read, or Goto, even though this is not good programming practice. The only exception is the keyword PARAMETER. If you plan to use variable names beginning with PARAMETER in an assignment statement, you need to use compiler option altparam.

Using keyword names for variables makes programs harder to read and understand. For readability, and to reduce the possibility of hard-to-find bugs, avoid using names that look like parts of Fortran statements. Rules that describe the context in which a keyword is recognized are discussed in Program Units and Procedures.

Argument keywords are a feature of Fortran 90 that let you specify dummy argument names when calling intrinsic procedures, or anywhere an interface (either implicit or explicit) is defined. Using argument keywords can make a program more readable and easy to follow. This is described more fully in Program Units and Procedures. The syntax statements in the A-Z Reference show the dummy keywords you can use for each Fortran procedure.

See Also

- Program Structure
- altparam compiler option

Character Sets

Intel Fortran supports the following characters:

- The Fortran 95/90 character set which consists of the following:
  - All uppercase and lowercase letters (A through Z and a through z)
  - The numerals 0 through 9
  - The underscore (_)
  - The following special characters:

Character	Name	Character	Name
blank or <Tab>	Blank (space) or tab	:	Colon
=	Equal sign	!	Exclamation point
+	Plus sign	"	Quotation mark
-	Minus sign	%	Percent sign
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left parenthesis	<	Less than
)	Right parenthesis	>	Greater than
,	Comma	?	Question mark
.	Period (decimal point)	$	Dollar sign (currency symbol)
'	Apostrophe		

- Other printable characters

  Printable characters include the tab character (09 hex), ASCII characters with codes in the range 20(hex) through 7E(hex), and characters in certain special character sets.

  Printable characters that are not in the Fortran 95/90 character set can only appear in comments, character constants, Hollerith constants, character string edit descriptors, and input/output records.

Uppercase and lowercase letters are treated as equivalent when used to specify program behavior (except in character constants and Hollerith constants).

See Also

- Program Structure, Characters, and Source Forms
- Data Types, Constants, and Variables
- ASCII and Key Code Charts for Windows*OS
- ASCII Character Set for Linux* OS and Mac OS* X

Source Forms

Within a program, source code can be in free, fixed, or tab form. Fixed or tab forms must not be mixed with free form in the same source program, but different source forms can be used in different source programs.

All source forms allow lowercase characters to be used as an alternative to uppercase characters.

Several characters are indicators in source code (unless they appear within a comment or a Hollerith or character constant). The following are rules for indicators in all source forms:

- Comment indicator

  A comment indicator can precede the first statement of a program unit and appear anywhere within a program unit. If the comment indicator appears within a source line, the comment extends to the end of the line.

  An all blank line is also a comment line.

  Comments have no effect on the interpretation of the program unit.

  For more information, see comment indicators in free source form, or fixed and tab source forms.

- Statement separator

  More than one statement (or partial statement) can appear on a single source line if a statement separator is placed between the statements. The statement separator is a semicolon character (;).

  Consecutive semicolons (with or without intervening blanks) are considered to be one semicolon.

  If a semicolon is the last character on a line, or the last character before a comment, it is ignored.

- Continuation indicator

  A statement can be continued for more than one line by placing a continuation indicator on the line. Intel Fortran allows at least 511 continuation lines for a fixed or tab source program and at least 255 continuation lines for a free form source program.

  Comments can occur within a continued statement, but comment lines cannot be continued.

  For more information, see continuation indicators in free source form, or fixed and tab source forms.

The following table summarizes characters used as indicators in source forms.

Table 523: Indicators in Source Forms

Source Item	Indicator [1]	Source Form	Position
Comment	!	All forms	Anywhere in source code
Comment line	!	Free	At the beginning of the source line
	!, C, or *	Fixed	In column 1
		Tab	In column 1
Continuation line [2]	&	Free	At the end of the source line
	Any character except zero or blank	Fixed	In column 6
	Any digit except zero	Tab	After the first tab
Statement separator	;	All forms	Between statements on the same line
Statement label	1 to 5 decimal digits	Free	Before a statement
		Fixed	In columns 1 through 5
		Tab	Before the first tab
A debugging statement[3]	D	Fixed	In column 1
		Tab	In column 1

[1] If the character appears in a Hollerith or character constant, it is not an indicator and is ignored.

[2] For fixed or tab source form, at least 511 continuation lines are allowed. For free source form, at least 255 continuation lines are allowed.

Source Item	Indicator [1]	Source Form	Position
[3] Fixed and tab forms only.			

Source form and line length can be changed at any time by using the FREEFORM, NOFREEFORM, or FIXEDFORMLINESIZE directives. The change remains in effect until the end of the file, or until changed again.

You can also select free source form by using compiler option free.

Source code can be written so that it is useable for all source forms.

Statement Labels

A *statement label* (or statement number) identifies a statement so that other statements can refer to it, either to get information or to transfer control. A label can precede any statement that is not part of another statement.

A statement label must be one to five decimal digits long; blanks and leading zeros are ignored. An all-zero statement label is invalid, and a blank statement cannot be labeled.

Labeled FORMAT and labeled executable statements are the only statements that can be referred to by other statement. FORMAT statements are referred to only in the format specifier of an I/O statement or in an ASSIGN statement. Two statements within a scoping unit cannot have the same label.

See Also

- Program Structure, Characters, and Source Forms
- Free Source Form
- Fixed and Tab Source Forms
- Source Code Useable for All Source Forms
- free compiler option

Free Source Form

In free source form, statements are not limited to specific positions on a source line. In Fortran 95/90, a free form source line can contain from 0 to 132 characters. Intel Fortran allows the line to be of any length.

Blank characters are significant in free source form. The following are rules for blank characters:

- Blank characters must not appear in lexical tokens, except within a character context. For example, there can be no blanks between the exponentiation operator **. Blank characters can be used freely between lexical tokens to improve legibility.

- Blank characters must be used to separate names, constants, or labels from adjacent keywords, names, constants, or labels. For example, consider the following statements:

```
INTEGER NUM

GO TO 40

20 DO K=1,8
```

  The blanks are required after INTEGER, TO, 20, and DO.

- Some adjacent keywords must have one or more blank characters between them. Others do not require any; for example, BLOCK DATA can also be spelled BLOCKDATA. The following list shows which keywords have optional or required blanks:

Optional Blanks	Required Blanks
BLOCK DATA	CASE DEFAULT
DOUBLE COMPLEX	DO WHILE
DOUBLE PRECISION	IMPLICIT *type-specifier*
ELSE IF	IMPLICIT NONE
ELSE WHERE	INTERFACE ASSIGNMENT
END BLOCK DATA	INTERFACE OPERATOR
END DO	MODULE PROCEDURE
END FILE	RECURSIVE FUNCTION
END FORALL	RECURSIVE SUBROUTINE
END FUNCTION	RECURSIVE *type-specifier* FUNCTION
END IF	*type-specifier* FUNCTION
END INTERFACE	*type-specifier* RECURSIVE FUNCTION
END MODULE	
END PROGRAM	
END SELECT	
END SUBROUTINE	

Optional Blanks	Required Blanks
END TYPE	
END WHERE	
GO TO	
IN OUT	
SELECT CASE	

For information on statement separators (;) in all forms, see Source Forms.

Comment Indicator

In free source form, the exclamation point character (!) indicates a comment if it is within a source line, or a comment line if it is the first character in a source line.

Continuation Indicator

In free source form, the ampersand character (&) indicates a continuation line (unless it appears in a Hollerith or character constant, or within a comment). The continuation line is the first noncomment line following the ampersand. Although Fortran 90 permits up to 39 continuation lines in free-form programs, Intel Fortran allows up to 511 continuation lines.

The following shows a continued statement:

```
TCOSH(Y) = EXP(Y) + &        ! The initial statement line
  EXP(-Y)           ! A continuation line
```

If the first nonblank character on the next noncomment line is an ampersand, the statement continues at the character following the ampersand. For example, the preceding example can be written as follows:

```
TCOSH(Y) = EXP(Y) + &
 & EXP(-Y)
```

If a lexical token must be continued, the first nonblank character on the next noncomment line must be an ampersand followed immediately by the rest of the token. For example:

```
   TCOSH(Y) = EXP(Y) + EX&
&P(-Y)
```

If you continue a character constant, an ampersand must be the first non-blank character of the continued line; the statement continues with the next character following the ampersand. For example:

```
ADVERTISER = "Davis, O'Brien, Chalmers & Peter&

        &son"

ARCHITECT  = "O'Connor, Emerson, and Davis&

      &  Associates"
```

If the ampersand is omitted on the continued line, the statement continues with the first non-blank character in the continued line. So, in the preceding example, the whitespace before "Associates" would be ignored.

The ampersand cannot be the only nonblank character in a line, or the only nonblank character before a comment; an ampersand in a comment is ignored.

Fixed and Tab Source Forms

In Fortran 95, fixed source form is identified as obsolescent.

In fixed and tab source forms, there are restrictions on where a statement can appear within a line.

By default, a statement can extend to character position 72. In this case, any text following position 72 is ignored and no warning message is printed. You can specify compiler option extend-source to extend source lines to character position 132.

Except in a character context, blanks are not significant and can be used freely throughout the program for maximum legibility.

Some Fortran compilers use blanks to pad short source lines out to 72 characters. By default, Intel Fortran does not. If portability is a concern, you can use the concatenation operator to prevent source lines from being padded by other Fortran compilers (see the example in "Continuation Indicator" below) or you can force short source lines to be padded by using compiler option pad-source.

Comment Indicator

In fixed and tab source forms, the exclamation point character (!) indicates a comment if it is within a source line. (It must not appear in column 6 of a fixed form line; that column is reserved for a continuation indicator.)

The letter C (or c), an asterisk (*), or an exclamation point (!) indicates a comment line when it appears in column 1 of a source line.

Continuation Indicator

In fixed and tab source forms, a continuation line is indicated by one of the following:

- For fixed form: Any character (except a zero or blank) in column 6 of a source line
- For tab form: Any digit (except zero) after the first tab

The compiler considers the characters following the continuation indicator to be part of the previous line. Although Fortran 95/90 permits up to 19 continuation lines in a fixed-form program, Intel Fortran allows up to 511 continuation lines.

If a zero or blank is used as a continuation indicator, the compiler considers the line to be an initial line of a Fortran statement.

The statement label field of a continuation line must be blank (except in the case of a debugging statement).

When long character or Hollerith constants are continued across lines, portability problems can occur. Use the concatenation operator to avoid such problems. For example:

```
      PRINT *, 'This is a very long character constant '//
     +         'which is safely continued across lines'
```

Use this same method when initializing data with long character or Hollerith constants. For example:

```
      CHARACTER*(*) LONG_CONST
      PARAMETER (LONG_CONST = 'This is a very long '//
     + 'character constant which is safely continued '//
     + 'across lines')
      CHARACTER*100 LONG_VAL
      DATA LONG_VAL /LONG_CONST/
```

Hollerith constants must be converted to character constants before using the concatenation method of line continuation.

The Fortran Standard requires that, within a program unit, the END statement cannot be continued, and no other statement in the program unit can have an initial line that appears to be the program unit END statement. In these instances, Intel Fortran produces warnings when standards checking is requested.

Debugging Statement Indicator

In fixed and tab source forms, the statement label field can contain a statement label, a comment indicator, or a debugging statement indicator.

The letter D indicates a debugging statement when it appears in column 1 of a source line. The initial line of the debugging statement can contain a statement label in the remaining columns of the statement label field.

If a debugging statement is continued onto more than one line, every continuation line must begin with a D and a continuation indicator.

By default, the compiler treats debugging statements as comments. However, you can specify compiler option d-lines to force the compiler to treat debugging statements as source text to be compiled.

Fixed-Format Lines

In fixed source form, a source line has columns divided into fields for statement labels, continuation indicators, statement text, and sequence numbers. Each column represents a single character.

The column positions for each field follow:

Field	Column
Statement label	1 through 5
Continuation indicator	6
Statement	7 through 72 (or 132 with compiler option extend-source)
Sequence number	73 through 80

By default, a sequence number or other identifying information can appear in columns 73 through 80 of any fixed-format line in an Intel Fortran program. The compiler ignores the characters in this field.

If you extend the statement field to position 132, the sequence number field does not exist.

NOTE. If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

See Also

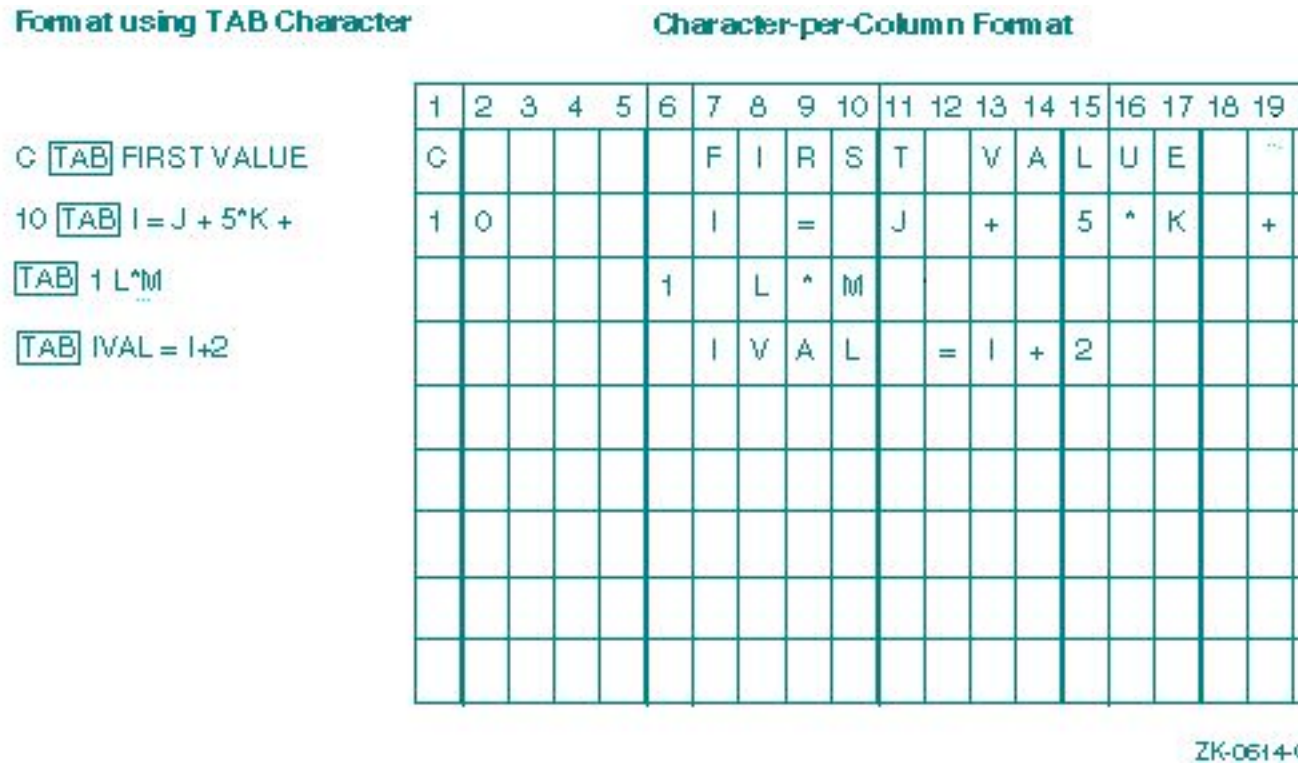
- Fixed and Tab Source Forms
- Source Forms
- Fixed and Tab Source Forms
- extend-source compiler option

Tab-Format Lines

In tab source form, you can specify a statement label field, a continuation indicator field, and a statement field, but not a sequence number field.

The following figure shows equivalent source lines coded with tab and fixed source form.

Line Formatting Example



The statement label field precedes the first tab character. The continuation indicator field and statement field follow the first tab character.

The continuation indicator is any nonzero digit. The statement field can contain any Fortran statement. A Fortran statement cannot start with a digit.

If a statement is continued, a continuation indicator must be the first character (following the first tab) on the continuation line.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you press the <Tab> key. However, the Intel Fortran compiler does not interpret the tab character in this way. It treats the tab character in a statement field the same way it treats a blank character. In the source listing that the compiler produces, the tab causes the character that follows to be printed at the next tab stop (usually located at columns 9, 17, 25, 33, and so on).

NOTE. If you use the sequence number field, do not use tabs anywhere in the source line, or the compiler may interpret the sequence numbers as part of the statement field in your program.

See Also

- Fixed and Tab Source Forms
- Source Forms
- Fixed and Tab Source Forms

Source Code Useable for All Source Forms

To write source code that is useable for all source forms (free, fixed, or tab), follow these rules:

Blanks	Treat as significant (see Free Source Form).
Statement labels	Place in column positions 1 through 5 (or before the first tab character).
Statements	Start in column position 7 (or after the first tab character).
Comment indicator	Use only !. Place anywhere *except* in column position 6 (or immediately after the first tab character).

Continuation indicator	Use only &. Place in column position 73 of the initial line and each continuation line, and in column 6 of each continuation line (no tab character can precede the ampersand in column 6).

The following example is valid for all source forms:

```
Column:
12345678...                                                              73
_____________________________________________________________________________
! Define the user function MY_SIN
      DOUBLE PRECISION FUNCTION MY_SIN(X)
        MY_SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)                      &
     &           - X**7/FACTOR(7)
      CONTAINS
        INTEGER FUNCTION FACTOR(N)
         FACTOR = 1
         DO 10 I = N, 1, -1
  10     FACTOR = FACTOR * I
        END FUNCTION FACTOR
      END FUNCTION MY_SIN
```

Data Types, Constants, and Variables

44

Each constant, variable, array, expression, or function reference in a Fortran statement has a data type. The data type of these items can be inherent in their construction, implied by convention, or explicitly declared.

Each *data type* has the following properties:

- A name

  The names of the intrinsic data types are predefined, while the names of derived types are defined in derived-type definitions. Data objects (constants, variables, or parts of constants or variables) are declared using the name of the data type.

- A set of associated values

  Each data type has a set of valid values. Integer and real data types have a range of valid values. Complex and derived types have sets of values that are combinations of the values of their individual components.

- A way to represent constant values for the data type

  A *constant* is a data object with a fixed value that cannot be changed during program execution. The value of a constant can be a numeric value, a logical value, or a character string.

  A constant that does not have a name is a *literal constant*. A literal constant must be of intrinsic type and it cannot be array-valued.

  A constant that has a name is a *named constant*. A named constant can be of any type, including derived type, and it can be array-valued. A named constant has the PARAMETER attribute and is specified in a type declaration statement or PARAMETER statement.

- A set of operations to manipulate and interpret these values

  The data type of a variable determines the operations that can be used to manipulate it. Besides intrinsic operators and operations, you can also define operators and operations.

Intrinsic Data Types

Intel® Fortran provides the following intrinsic data types:

- INTEGER

  There are four kind parameters for data of type integer:

  - INTEGER([KIND=]1) or INTEGER*1
  - INTEGER([KIND=]2) or INTEGER*2

- INTEGER([KIND=]4) or INTEGER*4
- INTEGER([KIND=]8) or INTEGER*8

- REAL

  There are three kind parameters for data of type real:

  - REAL([KIND=]4) or REAL*4
  - REAL([KIND=]8) or REAL*8
  - REAL([KIND=]16) or REAL*16

- DOUBLE PRECISION

  No kind parameter is permitted for data declared with type DOUBLE PRECISION. This data type is the same as REAL([KIND=]8).

- COMPLEX

  There are two kind parameters for data of type complex:

  - COMPLEX([KIND=]4) or COMPLEX*8
  - COMPLEX([KIND=]8) or COMPLEX*16
  - COMPLEX([KIND=]16) or COMPLEX*32

- DOUBLE COMPLEX

  No kind parameter is permitted for data declared with type DOUBLE COMPLEX. This data type is the same as COMPLEX([KIND=]8).

- LOGICAL

  There are four kind parameters for data of type logical:

  - LOGICAL([KIND=]1) or LOGICAL*1
  - LOGICAL([KIND=]2) or LOGICAL*2
  - LOGICAL([KIND=]4) or LOGICAL*4
  - LOGICAL([KIND=]8) or LOGICAL*8

- CHARACTER

  There is one kind parameter for data of type character: CHARACTER([KIND=]1).

- BYTE

  This is a 1-byte value; the data type is equivalent to INTEGER([KIND=]1).

The intrinsic function KIND can be used to determine the kind type parameter of a representation method.

For more portable programs, you should not use the forms INTEGER([KIND=]n) or REAL([KIND=]n). You should instead define a PARAMETER constant using the SELECTED_INT_KIND or SELECTED_REAL_KIND function, whichever is appropriate. For example, the following statements define a PARAMETER constant for an INTEGER kind that has 9 digits:

```
INTEGER, PARAMETER :: MY_INT_KIND = SELECTED_INT_KIND(9)
  ...
INTEGER(MY_INT_KIND) :: J
  ...
```

Note that the syntax :: is used in type declaration statements.

The following sections describe the intrinsic data types and forms for literal constants for each type.

See Also

- Data Types, Constants, and Variables
- Integer Data Types
- Real Data Types
- KIND
- Declaration Statements for Noncharacter Types
- Declaration Statements for Character Types
- Expressions
- Data Type Storage Requirements table

Integer Data Types

Integer data types can be specified as follows:

INTEGER

INTEGER([KIND=]*n*)

INTEGER**n*

n Is an initialization expression that evaluates to kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is default integer.

- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind that holds the constant.

Default integer is affected by compiler option integer-size, the INTEGER compiler directive, and the OPTIONS statement.

The intrinsic inquiry function KIND returns the kind type parameter, if you do not know it. You can use the intrinsic function SELECTED_INT_KIND to find the kind values that provide a given range of integer values. The decimal exponent range is returned by the intrinsic function RANGE.

Examples

The following examples show ways an integer variable can be declared.

An entity-oriented example is:

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

An attribute-oriented example is:

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

An integer can be used in certain cases when a logical value is expected, such as in a logical expression evaluating a condition, as in the following:

```
INTEGER I, X
READ (*,*) I
IF (I) THEN
  X = 1
END IF
```

Integer Constants

An *integer constant* is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.

Integer constants take the following form:

[*s*]*n*[*n*...][_*k*]

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). Any leading zeros are ignored.
k	Is the optional kind parameter: 1 for INTEGER(1), 2 for INTEGER(2), 4 for INTEGER(4), or 8 for INTEGER(8). It must be preceded by an underscore (_).

An unsigned constant is assumed to be nonnegative.

Integer constants are interpreted as decimal values (base 10) by default. To specify a constant that is not in base 10, use the following extension syntax:

[*s*] [[*base*] #] *nnn*...

s	Is an optional plus (+) or minus (-) sign.
base	Is any constant from 2 through 36. If *base* is omitted but # is specified, the integer is interpreted in base 16. If both *base* and # are omitted, the integer is interpreted in base 10. For bases 11 through 36, the letters A through Z represent numbers greater than 9. For example, for base 36, A represents 10, B represents 11, C represents 12, and so on, through Z, which represents 35. The case of the letters is not significant.

Note that compiler option integer-size can affect INTEGER data.

Examples

Table 527: Valid Integer (base 10) Constants

0
-127
+32123
47_2

Table 528: Invalid Integer (base 10) Constants

9999999999999999999	Number too large.
3.14	Decimal point not allowed; this is a valid REAL constant.

32,767	Comma not allowed.
33_3	3 is not a valid kind type for integers.

The following seven integers are all assigned a value equal to 3,994,575 decimal:

```
I = 2#11110011110001111001111
m = 7#45644664
J = +8#17171717
K = #3CF3CF
n = +17#2DE110
L = 3994575
index = 36#2DM8F
```

You can use integer constants to assign values to data. The following table shows assignments to different data and lists the integer and hexadecimal values in the data:

Fortran Assignment	Integer Value in Data	Hexadecimal Value in Data
LOGICAL(1)X		
INTEGER(1)X		
X = -128	-128	Z'80'
X = 127	127	Z'7F'
X = 255	-1	Z'FF'
LOGICAL(2)X		
INTEGER(2)X		
X = 255	255	Z'FF'
X = -32768	-32768	Z'8000'
X = 32767	32767	Z'7FFF'
X = 65535	-1	Z'FFFF'

See Also

- Integer Data Types
- Numeric Expressions
- integer-size compiler option

Building Applications for details on the ranges for integer types and kinds

Real Data Types

Real data types can be specified as follows:

REAL

REAL([KIND=]*n*)

REAL*_n_

DOUBLE PRECISION

n Is an initialization expression that evaluates to kind 4, 8, or 16.

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is default real.

Default real is affected by compiler options specifying real size and by the REAL directive.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size. If this compiler option is not specified, default DOUBLE PRECISION is REAL(8).

No kind parameter is permitted for data declared with type DOUBLE PRECISION.

The intrinsic inquiry function KIND returns the kind type parameter. The intrinsic inquiry function RANGE returns the decimal exponent range, and the intrinsic function PRECISION returns the decimal precision. You can use the intrinsic function SELECTED_REAL_KIND to find the kind values that provide a given precision and exponent range.

Examples

The following examples show how real variables can be declared.

An entity-oriented example is:

```
REAL (KIND = high), OPTIONAL :: testval

REAL, SAVE :: a(10), b(20,30)
```

An attribute-oriented example is:

```
REAL (KIND = high) testval

REAL a(10), b(20,30)

OPTIONAL testval

SAVE a, b
```

General Rules for Real Constants

A *real constant* approximates the value of a mathematical real number. The value of the constant can be positive, zero, or negative.

The following is the general form of a real constant with no exponent part:

[*s*]*n*[*n*...][_*k*]

A real constant with an exponent part has one of the following forms:

[*s*]*n*[*n*...]E[*s*]*nn*...[_*k*]

[*s*]*n*[*n*...]D[*s*]*nn*...

[*s*]*n*[*n*...]Q[*s*]*nn*...

s	Is a sign; required if negative (-), optional if positive (+).
n	Is a decimal digit (0 through 9). A decimal point must appear if the real constant has no exponent part.
k	Is the optional kind parameter: 4 for REAL(4), 8 for REAL(8), or 16 for REAL(16). It must be preceded by an underscore (_).

Description

Leading zeros (zeros to the left of the first nonzero digit) are ignored in counting significant digits. For example, in the constant 0.00001234567, all of the nonzero digits, and none of the zeros, are significant. (See the following sections for the number of significant digits each kind type parameter typically has).

The exponent represents a power of 10 by which the preceding real or integer constant is to be multiplied (for example, 1.0E6 represents the value 1.0 * 10**6).

A real constant with no exponent part and no kind type parameter is (by default) a single-precision (REAL(4)) constant. You can change the default behavior by specifying compiler option fpconstant.

If the real constant has no exponent part, a decimal point must appear in the string (anywhere before the optional kind parameter). If there is an exponent part, a decimal point is optional in the string preceding the exponent part; the exponent part must not contain a decimal point.

The exponent letter E denotes a single-precision real (REAL(4)) constant, unless the optional kind parameter specifies otherwise. For example, -9.E2_8 is a double-precision constant (which can also be written as -9.D2).

The exponent letter D denotes a double-precision real (REAL(8)) constant.

The exponent letter Q denotes a quad-precision real (REAL(16)) constant.

A minus sign must appear before a negative real constant; a plus sign is optional before a positive constant. Similarly, a minus sign must appear between the exponent letter (E, D, or Q) and a negative exponent, whereas a plus sign is optional between the exponent letter and a positive exponent.

If the real constant includes an exponent letter, the exponent field cannot be omitted, but it can be zero.

To specify a real constant using both an exponent letter and a kind parameter, the exponent letter must be E, and the kind parameter must follow the exponent part.

See Also

- Real Data Types
- fpconstant compiler option

REAL(4) Constants

A single-precision REAL constant occupies four bytes of memory. The number of digits is unlimited, but typically only the leftmost seven digits are significant.

IEEE* S_floating format is used.

Note that compiler option real-size can affect REAL data.

Examples

Table 529: Valid REAL(4) Constants

```
3.14159

3.14159_4

621712._4

-.00127

+5.0E3

2E-3_4
```

Table 530: Invalid REAL(4) Constants

`1,234,567.`	Commas not allowed.
`325E-47`	Too small for REAL; this is a valid DOUBLE PRECISION constant.

`-47.E47`	Too large for REAL; this is a valid DOUBLE PRECISION constant.
`625._6`	6 is not a valid kind for reals.
`100`	Decimal point missing; this is a valid integer constant.
`$25.00`	Special character not allowed.

See Also

- Real Data Types
- General Rules for Real Constants
- real-size compiler option

Building Applications for details on the format and range of REAL(4) data

REAL(8) or DOUBLE PRECISION Constants

A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.

A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.

IEEE T_floating format is used.

Note that compiler option double-size can affect DOUBLE PRECISION data.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size.

Examples

Table 531: Valid REAL(8) or DOUBLE PRECISION Constants

```
123456789D+5

123456789E+5_8

+2.7843D00

-.522D-12

2E200_8
```

2.3_8
3.4E7_8

Table 532: Invalid REAL(8) or DOUBLE PRECISION Constants

-.25D0_2	2 is not a valid kind for reals.
+2.7182812846182	No D exponent designator is present; this is a valid single-precision constant.
123456789.D400	Too large for any double-precision format.
123456789.D-400	Too small for any double-precision format.

See Also

- Real Data Types
- General Rules for Real Constants
- -double-size compiler option

Building Applications for details on the format and range of DOUBLE PRECISION (REAL(8)) data

REAL(16) Constants

A REAL(16) constant has more than four times the accuracy of a REAL(4) number, and a greater range.

A REAL(16) constant occupies 16 bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 33 digits are significant.

IEEE X_floating format is used.

Examples

Table 533: Valid REAL(16) Constants

123456789Q4000
-1.23Q-400
+2.72Q0
1.88_16

Table 534: Invalid REAL(16) Constants

`1.Q5000`	Too large.
`1.Q-5000`	Too small.

See Also

- Real Data Types
- General Rules for Real Constants

Building Applications for details on the format and range of REAL(16) data

Complex Data Types

Complex data types can be specified as follows:

COMPLEX

COMPLEX([KIND=]*n*)

COMPLEX**s*

DOUBLE COMPLEX

n	Is an initialization expression that evaluates to kind 4, 8, or 16.
s	Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type default complex.

Default real is affected by compiler option real-size and by the REAL directive.

The default KIND for DOUBLE COMPLEX is affected by compiler option `double-size`. If the compiler option is not specified, default DOUBLE COMPLEX is COMPLEX(8).

No kind parameter is permitted for data declared with type DOUBLE COMPLEX.

Examples

The following examples show how complex variables can be declared.

An entity-oriented example is:

```
COMPLEX (4), DIMENSION (8) :: cz, cq
```

An attribute-oriented example is:

```
COMPLEX(4) cz, cq
DIMENSION(8) cz, cq
```

General Rules for Complex Constants

A *complex constant* approximates the value of a mathematical complex number. The constant is a pair of real or integer values, separated by a comma, and enclosed in parentheses. The first constant represents the real part of that number; the second constant represents the imaginary part.

The following is the general form of a complex constant:

(*c*,*c*)

c — Is as follows:

- For COMPLEX(4) constants, *c* is an integer or REAL(4) constant.
- For COMPLEX(8) constants, *c* is an integer, REAL(4) constant, or DOUBLE PRECISION (REAL(8)) constant. At least one of the pair must be DOUBLE PRECISION.
- For COMPLEX(16) constants, *c* is an integer, REAL(4) constant, REAL(8) constant, or REAL(16) constant. At least one of the pair must be a REAL(16) constant.

Note that the comma and parentheses are required.

COMPLEX(4) Constants

A COMPLEX(4) constant is a pair of integer or single-precision real constants that represent a complex number.

A COMPLEX(4) constant occupies eight bytes of memory and is interpreted as a complex number.

If the real and imaginary part of a complex literal constant are both real, the kind parameter value is that of the part with the greater decimal precision.

The rules for REAL(4) constants apply to REAL(4) constants used in COMPLEX constants. (See General Rules for Complex Constants and REAL(4) Constants for the rules on forming REAL(4) constants.)

The REAL(4) constants in a COMPLEX constant have IEEE S_floating format.

Note that compiler option real-size can affect REAL data.

Examples

Table 535: Valid COMPLEX(4) Constants

```
(1.7039,-1.70391)

(44.36_4,-12.2E16_4)

(+12739E3,0.)

(1,2)
```

Table 536: Invalid COMPLEX(4) Constants

`(1.23,)`	Missing second integer or single-precision real constant.
`(1.0, 2H12)`	Hollerith constant not allowed.

See Also

- Complex Data Types
- General Rules for Complex Constants
- real-size compiler option

Building Applications for details on the format and range of COMPLEX (COMPLEX(4)) data

COMPLEX(8) or DOUBLE COMPLEX Constants

A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(8) or DOUBLE COMPLEX constants. (See General Rules for Complex Constants and REAL(8) or DOUBLE PRECISION Constants for the rules on forming DOUBLE PRECISION constants.)

The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE T_floating format.

The default KIND for DOUBLE COMPLEX is affected by compiler option double-size.

Examples

Table 537: Valid COMPLEX(8) or DOUBLE COMPLEX Constants

```
(1.7039,-1.7039D0)

(547.3E0_8,-1.44_8)

(1.7039E0,-1.7039D0)

(+12739D3,0.D0)
```

Table 538: Invalid COMPLEX(8) or DOUBLE COMPLEX Constants

`(1.23D0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1,1.2)`	Neither constant is DOUBLE PRECISION; this is a valid single-precision constant.

See Also

- Complex Data Types
- General Rules for Complex Constants
- -double-size

Building Applications for details on the format and range of DOUBLE COMPLEX data

COMPLEX(16) Constants

A COMPLEX(16) constant is a pair of constants that represents a complex number. One of the pair must be a REAL(16) constant, the other can be an integer, single-precision real, double-precision real, or REAL(16) constant.

A COMPLEX(16) constant occupies 32 bytes of memory and is interpreted as a complex number.

The rules for REAL(16) constants apply to REAL(16) constants used in COMPLEX constants. (See General Rules for Complex Constants and REAL(16) Constants for the rules on forming REAL(16) constants.)

The REAL(16) constants in a COMPLEX constant have IEEE X_floating format.

Note that compiler option real-size can affect REAL data.

Examples

Table 539: Valid COMPLEX(16) Constants

```
(1.7039,-1.7039Q2)

(547.3E0_16,-1.44)

(+12739D3,0.Q0)
```

Table 540: Invalid COMPLEX(16) Constants

`(1.23Q0,)`	Second constant missing.
`(1D1,2H12)`	Hollerith constants not allowed.
`(1.7039E0,-1.7039D0)`	Neither constant is REAL(16); this is a valid double-precision constant.

See Also

- Complex Data Types
- General Rules for Complex Constants
- real-size compiler option

Building Applications for details on the format and range of COMPLEX(16) data

Logical Data Types

Logical data types can be specified as follows:

LOGICAL

LOGICAL([KIND=]*n*)

LOGICAL*_n_

n Is an initialization expression that evaluates to kind 1, 2, 4, or 8.

If a kind parameter is specified, the logical constant has the kind specified. If no kind parameter is specified, the kind of the constant is default logical.

Examples

The following examples show how logical variables can be declared.

An entity-oriented example is:

```
LOGICAL, ALLOCATABLE :: flag1, flag2

LOGICAL (KIND = byte), SAVE :: doit, dont
```

An attribute-oriented example is:

```
LOGICAL flag1, flag2

LOGICAL (KIND = byte) doit, dont

ALLOCATABLE flag1, flag2

SAVE doit, dont
```

Logical Constants

A logical constant represents only the logical values true or false, and takes one of the following forms:

.TRUE.[_*k*]

.FALSE.[_*k*]

k	Is the optional kind parameter: 1 for LOGICAL(1), 2 for LOGICAL(2), 4 for LOGICAL(4), or 8 for LOGICAL(8). It must be preceded by an underscore (_).

The numeric value of .TRUE. and .FALSE. can be -1 and 0 or 1 and 0 depending on compiler option fpscomp [no]logicals. Logical data can take on integer data values. Logical data type ranges correspond to their comparable integer data type ranges. For example, the LOGICAL(2) range is the same as the INTEGER(2) range.

See Also

- Logical Data Types

Building Applications for details on integer data type ranges

Character Data Type

The character data type can be specified as follows:

CHARACTER

CHARACTER([LEN=] *len*)

CHARACTER(LEN= *len*, KIND= *n*)

CHARACTER(len, [KIND=] *n*)

CHARACTER(KIND= *n* [, LEN= *len*])

CHARACTER* *len* [,]

n	Is an initialization expression that evaluates to kind 1.
len	Is a string length (not a kind). For more information, see Declaration Statements for Character Types.

If no kind type parameter is specified, the kind of the constant is default character.

On Windows systems, several Multi-Byte Character Set (MBCS) functions are available to manipulate special non-English characters.

Character Constants

A *character constant* is a character string enclosed in delimiters (apostrophes or quotation marks). It takes one of the following forms:

[*k*_]'[*ch*...]' [*C*]

[*k*_]"[*ch*...]" [*C*]

k	Is the optional kind parameter: 1 (the default). It must be followed by an underscore (_). Note that in character constants, the kind must precede the constant.
ch	Is an ASCII character.
C	Is a C string specifier. C strings can be used to define strings with nonprintable characters. For more information, see C Strings in Character Constants.

Description

The value of a character constant is the string of characters between the delimiters. The value does not include the delimiters, but does include all blanks or tabs within the delimiters.

If a character constant is delimited by apostrophes, use two consecutive apostrophes ('') to place an apostrophe character in the character constant.

Similarly, if a character constant is delimited by quotation marks, use two consecutive quotation marks ("") to place a quotation mark character in the character constant.

The length of the character constant is the number of characters between the delimiters, but two consecutive delimiters are counted as one character.

The length of a character constant must be in the range of 0 to 7188. Each character occupies one byte of memory.

If a character constant appears in a numeric context (such as an expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant.

A zero-length character constant is represented by two consecutive apostrophes or quotation marks.

Examples

Table 541: Valid Character Constants

`"WHAT KIND TYPE? "`
`'TODAY''S DATE IS: '`
`"The average is: "`
`''`

Table 542: Invalid Character Constants

`'HEADINGS`	No trailing apostrophe.
`'Map Number:"`	Beginning delimiter does not match ending delimiter.

See Also

- Character Data Type
- Declaration Statements for Character Types

C Strings in Character Constants

String values in the C language are terminated with null characters (CHAR(0)) and can contain nonprintable characters (such as backspace).

Nonprintable characters are specified by escape sequences. An escape sequence is denoted by using the backslash (\) as an escape character, followed by a single character indicating the nonprintable character desired.

This type of string is specified by using a standard string constant followed by the character C. The standard string constant is then interpreted as a C-language constant. Backslashes are treated as escapes, and a null character is automatically appended to the end of the string (even if the string already ends in a null character).

The following table shows the escape sequences that are allowed in character constants:

Table 543: C-Style Escape Sequences

Escape Sequence	Represents
\a or \A	A bell
\b or \B	A backspace
\f or \F	A formfeed
\n or \N	A new line
\r or \R	A carriage return
\t or \T	A horizontal tab
\v or \V	A vertical tab
\x*hh* or \X*hh*	A hexadecimal bit pattern
ooo	An octal bit pattern
\0	A null character
\\	A backslash

If a string contains an escape sequence that isn't in this table, the backslash is ignored.

A C string must also be a valid Fortran string. If the string is delimited by apostrophes, apostrophes in the string itself must be represented by two consecutive apostrophes ('').

For example, the escape sequence `\'string` causes a compiler error because Fortran interprets the apostrophe as the end of the string. The correct form is `\''string`.

If the string is delimited by quotation marks, quotation marks in the string itself must be represented by two consecutive quotation marks ("").

The sequences \ooo and \x*hh* allow any ASCII character to be given as a one- to three-digit octal or a one- to two-digit hexadecimal character code. Each octal digit must be in the range 0 to 7, and each hexadecimal digit must be in the range 0 to F. For example, the C strings `'\010'C` and `'\x08'C` both represent a backspace character followed by a null character.

The C string `'\\abcd'C` is equivalent to the string `'\abcd'` with a null character appended. The string `''C` represents the ASCII null character.

Character Substrings

A *character substring* is a contiguous segment of a character string. It takes one of the following forms:

v ([*e1*]:[*e2*])

a (*s* [, *s*] . . .) ([*e1*]:[*e2*])

v	Is a character scalar constant, or the name of a character scalar variable or character structure component.
e1	Is a scalar integer (or other numeric) expression specifying the leftmost character position of the substring; the *starting* point.
e2	Is a scalar integer (or other numeric) expression specifying the rightmost character position of the substring; the *ending* point.
a	Is the name of a character array.
s	Is a subscript expression.

Both *e1* and *e2* must be within the range 1,2, ..., *len*, where *len* is the length of the parent character string. If *e1* exceeds *e2*, the substring has length zero.

Description

Character positions within the parent character string are numbered from left to right, beginning at 1.

If the value of the numeric expression *e1* or *e2* is not of type integer, it is converted to integer before use (any fractional parts are truncated).

If *e1* is omitted, the default is 1. If *e2* is omitted, the default is *len*. For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

Examples

Consider the following example:

```
CHARACTER*8 C, LABEL

LABEL = 'XVERSUSY'

C = LABEL(2:7)
```

LABEL(2:7) specifies the substring starting with the second character position and ending with the seventh character position of the character variable assigned to LABEL, so C has the value `'VERSUS'`.

Consider the following example:

```
TYPE ORGANIZATION
  INTEGER ID
  CHARACTER*35 NAME
END TYPE ORGANIZATION
TYPE(ORGANIZATION) DIRECTOR
CHARACTER*25 BRANCH, STATE(50)
```

The following are valid substrings based on this example:

```
BRANCH(3:15)           ! parent string is a scalar variable
STATE(20) (1:3)        ! parent string is an array element
DIRECTOR%NAME(:)       ! parent string is a structure component
```

Consider the following example:

```
CHARACTER(*), PARAMETER :: MY_BRANCH = "CHAPTER 204"
CHARACTER(3) BRANCH_CHAP
BRANCH_CHAP = MY_BRANCH(9:11)   ! parent string is a character constant
```

BRANCH_CHAP is a character string of length 3 that has the value `'204'`.

See Also

- Character Data Type
- Arrays
- Array Elements
- Structure Components

Derived Data Types

You can create derived data types from intrinsic data types or previously defined derived types.

A derived type is resolved into "ultimate" components that are either of intrinsic type or are pointers.

The set of values for a specific derived type consists of all possible sequences of component values permitted by the definition of that derived type. Structure constructors are used to specify values of derived types.

Nonintrinsic assignment for derived-type entities must be defined by a subroutine with an ASSIGNMENT interface. Any operation on derived-type entities must be defined by a function with an OPERATOR interface. Arguments and function values can be of any intrinsic or derived type.

See Also

- Data Types, Constants, and Variables
- Derived-Type Definition
- Default Initialization
- Structure Components
- Structure Constructors
- Derived-Type Definition
- Default Initialization
- Structure Components
- Structure Constructors
- Derived-Type Assignment Statements
- Defining Generic Operators
- Defining Generic Assignment
- Records

Derived-Type Definition

A derived-type definition specifies the name of a user-defined type and the types of its components.

See Also

- Derived Data Types
- TYPE

Default Initialization

Default initialization occurs if initialization appears in a derived-type component definition. (This is a Fortran 95 feature.)

The specified initialization of the component will apply even if the definition is PRIVATE.

Default initialization applies to dummy arguments with INTENT(OUT). It does not imply the derived-type component has the SAVE attribute.

Explicit initialization in a type declaration statement overrides default initialization.

To specify default initialization of an array component, use a constant expression that includes one of the following:

- An array constructor
- A single scalar that becomes the value of each array element

Pointers can have an association status of associated, disassociated, or undefined. If no default initialization status is specified, the status of the pointer is undefined. To specify disassociated status for a pointer component, use =>NULL( ).

Examples

You do not have to specify initialization for each component of a derived type. For example:

```
TYPE REPORT
  CHARACTER (LEN=20) REPORT_NAME
  INTEGER DAY
  CHARACTER (LEN=3) MONTH
  INTEGER :: YEAR = 1995        ! Only component with default
END TYPE REPORT                 !      initialization
```

Consider the following:

```
TYPE (REPORT), PARAMETER :: NOV_REPORT = REPORT ("Sales", 15, "NOV", 1996)
```

In this case, the explicit initialization in the type declaration statement overrides the YEAR component of NOV_REPORT.

The default initial value of a component can also be overridden by default initialization specified in the type definition. For example:

```
TYPE MGR_REPORT
  TYPE (REPORT) :: STATUS = NOV_REPORT
  INTEGER NUM
END TYPE MGR_REPORT
TYPE (MGR_REPORT) STARTUP
```

In this case, the STATUS component of STARTUP gets its initial value from NOV_REPORT, overriding the initialization for the YEAR component.

Structure Components

A reference to a component of a derived-type structure takes the following form:

parent [%*component* [(*s-list*)]] ... %*component* [(*s-list*)]

parent	Is the name of a scalar or array of derived type. The percent sign (%) is called a component selector.
component	Is the name of a component of the immediately preceding parent or component.
s-list	Is a list of one or more subscripts. If the list contains subscript triplets or vector subscripts, the reference is to an array section. Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension. The number of subscripts in any *s-list* must equal the rank of the immediately preceding parent or component.

Description

Each parent or component (except the rightmost) must be of derived type.

The parent or one of the components can have nonzero rank (be an array). Any component to the right of a parent or component of nonzero rank must not have the POINTER attribute.

The rank of the structure component is the rank of the part (parent or component) with nonzero rank (if any); otherwise, the rank is zero. The type and type parameters (if any) of a structure component are those of the rightmost part name.

The structure component must not be referenced or defined before the declaration of the parent object.

If the parent object has the INTENT, TARGET, or PARAMETER attribute, the structure component also has the attribute.

Examples

The following example shows a derived-type definition with two components:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

The following shows how to declare CONTRACT to be of type EMPLOYEE:

```
TYPE(EMPLOYEE) :: CONTRACT
```

Note that both examples started with the keyword TYPE. The first (initial) statement of a derived-type definition is called a derived-type statement, while the statement that declares a derived-type object is called a TYPE statement.

The following example shows how to reference component ID of parent structure CONTRACT:

```
CONTRACT%ID
```

The following example shows a derived type with a component that is a previously defined type:

```
TYPE DOT
  REAL X, Y
END TYPE DOT
....
TYPE SCREEN
  TYPE(DOT) C, D
END TYPE SCREEN
```

The following declares a variable of type SCREEN:

```
TYPE(SCREEN) M
```

Variable M has components M%C and M%D (both of type DOT); M%C has components M%C%X and M%C%Y of type REAL.

The following example shows a derived type with a component that is an array:

```
TYPE CAR_INFO
  INTEGER YEAR
  CHARACTER(LEN=15), DIMENSION(10) :: MAKER
  CHARACTER(LEN=10) MODEL, BODY_TYPE*8
  REAL PRICE
END TYPE
...
TYPE(CAR_INFO) MY_CAR
```

Note that MODEL has a character length of 10, but BODY_TYPE has a character length of 8. You can assign a value to a component of a structure; for example:

```
MY_CAR%YEAR = 1985
```

The following shows an array structure component:

```
MY_CAR%MAKER
```

In the preceding example, if a subscript list (or substring) was appended to MAKER, the reference would not be to an array structure component, but to an array element or section.

Consider the following:

```
MY_CAR%MAKER(2) (4:10)
```

In this case, the component is substring 4 to 10 of the second element of array MAKER.

Consider the following:

```
TYPE CHARGE
  INTEGER PARTS(40)
  REAL LABOR
  REAL MILEAGE
END TYPE CHARGE
TYPE(CHARGE) MONTH
TYPE(CHARGE) YEAR(12)
```

Some valid array references for this type follow:

```
MONTH%PARTS(I)           ! An array element
MONTH%PARTS(I:K)         ! An array section
YEAR(I)%PARTS            ! An array structure component (a whole array)
YEAR(J)%PARTS(I)         ! An array element
YEAR(J)%PARTS(I:K)       ! An array section
YEAR(J:K)%PARTS(I)       ! An array section
YEAR%PARTS(I)            ! An array section
```

The following example shows a derived type with a pointer component that is of the type being defined:

```
TYPE NUMBER
  INTEGER NUM
  TYPE(NUMBER), POINTER :: START_NUM => NULL( )
  TYPE(NUMBER), POINTER :: NEXT_NUM => NULL( )
END TYPE
```

A type such as this can be used to construct linked lists of objects of type NUMBER. Note that the pointers are given the default initialization status of disassociated.

The following example shows a private type:

```
TYPE, PRIVATE :: SYMBOL
  LOGICAL TEST
  CHARACTER(LEN=50) EXPLANATION
END TYPE SYMBOL
```

This type is private to the module. The module can be used by another scoping unit, but type SYMBOL is not available.

See Also

- Derived Data Types
- Array Elements
- Array Sections
- Modules and Module Procedures

Structure Constructors

A structure constructor lets you specify scalar values of a derived type. It takes the following form:

d-name (*expr-list*)

d-name	Is the name of the derived type.
expr-list	Is a list of expressions specifying component values. The values must agree in number and order with the components of the derived type. If necessary, values are converted (according to the rules of assignment), to agree with their corresponding components in type and kind parameters.

Description

A structure constructor must not appear before its derived type is defined.

If a component of the derived type is an array, the shape in the expression list must conform to the shape of the component array.

If a component of the derived type is a pointer, the value in the expression list must evaluate to an object that would be a valid target in a pointer assignment statement. (A constant is not a valid target in a pointer assignment statement.)

If all the values in a structure constructor are constant expressions, the constructor is a derived-type constant expression.

Examples

Consider the following derived-type definition:

```
TYPE EMPLOYEE
  INTEGER ID
  CHARACTER(LEN=40) NAME
END TYPE EMPLOYEE
```

This can be used to produce the following structure constructor:

```
EMPLOYEE(3472, "John Doe")
```

The following example shows a type with a component of derived type:

```
TYPE ITEM
  REAL COST
  CHARACTER(LEN=30) SUPPLIER
  CHARACTER(LEN=20) ITEM_NAME
END TYPE ITEM
TYPE PRODUCE
  REAL MARKUP
  TYPE(ITEM) FRUIT
END TYPE PRODUCE
```

In this case, you must use an embedded structure constructor to specify the values of that component; for example:

```
PRODUCE(.70, ITEM (.25, "Daniels", "apple"))
```

See Also

- Derived Data Types
- Pointer Assignments

Binary, Octal, Hexadecimal, and Hollerith Constants

Binary, octal, hexadecimal, and Hollerith constants are nondecimal constants. They have no intrinsic data type, but assume a numeric data type depending on their use.

Fortran 95/90 allows unsigned binary, octal, and hexadecimal constants to be used in DATA statements; the constant must correspond to an integer scalar variable.

In Intel Fortran, binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

Binary Constants

A *binary constant* is an alternative way to represent a numeric constant. A binary constant takes one of the following forms:

B'*d*[*d*...]'

B"*d*[*d*...]"

d	Is a binary (base 2) digit (0 or 1).

You can specify up to 128 binary digits in a binary constant.

Examples

Table 544: Valid Binary Constants

B'0101110'
B"1"

Table 545: Invalid Binary Constants

B'0112'	The character 2 is invalid.
B10011'	No apostrophe after the B.
"1000001"	No B before the first quotation mark.

See Also

- Binary, Octal, Hexadecimal, and Hollerith Constants
- Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Octal Constants

An *octal constant* is an alternative way to represent numeric constants. An octal constant takes one of the following forms:

O'*d*[*d*...]'

O"*d*[*d*...]"

d	Is an octal (base 8) digit (0 through 7).

You can specify up to 128 bits (43 octal digits) in octal constants.

Examples

Table 546: Valid Octal Constants

`O'07737'`
`O"1"`

Table 547: Invalid Octal Constants

`O'7782'`	The character 8 is invalid.
`O7772'`	No apostrophe after the O.
`"0737"`	No O before the first quotation mark.

See Also

- Binary, Octal, Hexadecimal, and Hollerith Constants
- Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Hexadecimal Constants

A *hexadecimal constant* is an alternative way to represent numeric constants. A hexadecimal constant takes one of the following forms:

Z'*d*[*d*...]'

Z"*d*[*d*...]"

d	Is a hexadecimal (base 16) digit (0 through 9, or an uppercase or lowercase letter in the range of A to F).

You can specify up to 128 bits (32 hexadecimal digits) in hexadecimal constants.

Examples

Table 548: Valid Hexadecimal Constants

```
Z'AF9730'

Z"FFABC"

Z'84'
```

Table 549: Invalid Hexadecimal Constants

`Z'999.'`	Decimal not allowed.
`ZF9"`	No quotation mark after the Z.

See Also

- Binary, Octal, Hexadecimal, and Hollerith Constants
- Alternative Syntax for Binary, Octal, and Hexadecimal Constants

Hollerith Constants

A *Hollerith constant* is a string of printable ASCII characters preceded by the letter H. Before the H, there must be an unsigned, nonzero default integer constant stating the number of characters in the string (including blanks and tabs).

Hollerith constants are strings of 1 to 2000 characters. They are stored as byte strings, one character per byte.

Examples

Table 550: Valid Hollerith Constants

```
16HTODAY'S DATE IS:

1HB

4H ABC
```

Table 551: Invalid Hollerith Constants

`3HABCD`	Wrong number of characters.
`0H`	Hollerith constants must contain at least one character.

Determining the Data Type of Nondecimal Constants

Binary, octal, hexadecimal, and Hollerith constants have no intrinsic data type. In most cases, the default integer data type is assumed.

However, these constants can assume a numeric data type depending on their use. When the constant is used with a binary operator (including the assignment operator), the data type of the constant is the data type of the other operand. For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`INTEGER(2) ICOUNT`		
`INTEGER(4) JCOUNT`		
`INTEGER(4) N`		
`REAL(8) DOUBLE`		
`REAL(4) RAFFIA, RALPHA`		
`RAFFIA = B'1001100111111010011'`	REAL(4)	4
`RAFFIA = Z'99AF2'`	REAL(4)	4
`RALPHA = 4HABCD`	REAL(4)	4
`DOUBLE = B'11111111111100110011010'`	REAL(8)	8
`DOUBLE = Z'FFF99A'`	REAL(8)	8
`DOUBLE = 8HABCDEFGH`	REAL(8)	8

Statement	Data Type of Constant	Length of Constant (in bytes)
`JCOUNT = ICOUNT + B'011101110111'`	INTEGER(2)	2
`JCOUNT = ICOUNT + O'777'`	INTEGER(2)	2
`JCOUNT = ICOUNT + 2HXY`	INTEGER(2)	2
`IF (N .EQ. B'1010100') GO TO 10`	INTEGER(4)	4
`IF (N .EQ. O'123') GO TO 10`	INTEGER(4)	4
`IF (N. EQ. 1HZ) GO TO 10`	INTEGER(4)	4

When a specific data type (generally integer) is required, that type is assumed for the constant. For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`Y(IX) = Y(O'15') + 3.`	INTEGER(4)	4
`Y(IX) = Y(1HA) + 3.`	INTEGER(4)	4

When a nondecimal constant is used as an actual argument, the following occurs:

- For binary, octal, and hexadecimal constants, if the value fits in a default integer, that integer kind is used. Otherwise, the smallest integer kind large enough to hold the value is used.
- For Hollerith constants, a numeric data type of sufficient size to hold the length of the constant is assumed.

For example:

Statement	Data Type of Constant	Length of Constant (in bytes)
`CALL APAC(Z'34BC2')`	INTEGER(4)	4

Statement	Data Type of Constant	Length of Constant (in bytes)
`CALL APAC(9HABCDEFGHI)`	REAL(16)	9

When a binary, octal, or hexadecimal constant is used in any other context, the default integer data type is assumed. In the following examples, default integer is INTEGER(4):

Statement	Data Type of Constant	Length of Constant (in bytes)
`IF (Z'AF77') 1,2,3`	INTEGER(4)	4
`IF (2HAB) 1,2,3`	INTEGER(4)	4
`I = O'7777' - Z'A39'` [1]	INTEGER(4)	4
`I = 1HC - 1HA`	INTEGER(4)	4
`J = .NOT. O'73777'`	INTEGER(4)	4
`J = .NOT. 1HB`	INTEGER(4)	4

[1] When two typeless constants are used in an operation, they both take default integer type.

When nondecimal constants are not the same length as the length implied by a data type, the following occurs:

- Binary, octal, and hexadecimal constants

  These constants can specify up to 16 bytes of data. When the length of the constant is less than the length implied by the data type, the leftmost digits have a value of zero.

  When the length of the constant is greater than the length implied by the data type, the constant is truncated on the left. An error results if any nonzero digits are truncated.

  The Data Type Storage Requirements table lists the number of bytes that each data type requires.

- Hollerith constants

  When the length of the constant is less than the length implied by the data type, blanks are appended to the constant on the right.

  When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. If any characters other than blank characters are truncated, a warning occurs.

Each Hollerith character occupies one byte of memory.

Variables

A variable is a data object whose value can be changed at any point in a program. A variable can be any of the following:

- A scalar

  A *scalar* is a single object that has a single value; it can be of any intrinsic or derived (user-defined) type.

- An array

  An *array* is a collection of scalar elements of any intrinsic or derived type. All elements must have the same type and kind parameters.

- A subobject designator

  A subobject is part of an object. The following are subobjects:

  An array element

  An array section

  A structure component

  A character substring

  For example, B(3) is a subobject (array element) designator for array B. A subobject cannot be a variable if its parent object is a constant.

The name of a variable is associated with a single storage location.

Variables are classified by data type, as constants are. The data type of a variable indicates the type of data it contains, including its precision, and implies its storage requirements. When data of any type is assigned to a variable, it is converted to the data type of the variable (if necessary).

A variable is *defined* when you give it a value. A variable can be defined before program execution by a DATA statement or a type declaration statement. During program execution, variables can be defined or redefined in assignment statements and input statements, or undefined (for example, if an I/O error occurs). When a variable is undefined, its value is unpredictable.

When a variable becomes undefined, all variables associated by storage association also become undefined.

An object with subobjects, such as an array, can only be defined when all of its subobjects are defined. Conversely, when at least one of its subobjects are undefined, the object itself, such as an array or derived type, is undefined.

This section also discusses the Data Types of Scalar Variables and Arrays.

Data Types of Scalar Variables

The data type of a scalar variable can be explicitly declared in a type declaration statement. If no type is declared, the variable has an implicit data type based on predefined typing rules or definitions in an IMPLICIT statement.

An explicit declaration of data type takes precedence over any implicit type. Implicit type specified in an IMPLICIT statement takes precedence over predefined typing rules.

Specification of Data Type

Type declaration statements explicitly specify the data type of scalar variables. For example, the following statements associate VAR1 with an 8-byte complex storage location, and VAR2 with an 8-byte double-precision storage location:

```
COMPLEX(8) VAR1

REAL(8) VAR2
```

NOTE. If no kind parameter is specified for a data type, the default kind is used. The default kind can be affected by compiler options that affect the size of variables.

You can explicitly specify the data type of a scalar variable only once.

If no explicit data type specification appears, any variable with a name that begins with the letter in the range specified in the IMPLICIT statement becomes the data type of the variable.

Character type declaration statements specify that given variables represent character values with the length specified. For example, the following statements associate the variable names INLINE, NAME, and NUMBER with storage locations containing character data of lengths 72, 12, and 9, respectively:

```
CHARACTER*72 INLINE

CHARACTER NAME*12, NUMBER*9
```

In single subprograms, assumed-length character arguments can be used to process character strings with different lengths. The assumed-length character argument has its length specified with an asterisk, for example:

```
CHARACTER*(*) CHARDUMMY
```

The argument CHARDUMMY assumes the length of the actual argument.

See Also

- Data Types of Scalar Variables
- Type declaration statements
- Assumed-length character arguments
- IMPLICIT statement
- Declaration Statements for Character Types

Implicit Typing Rules

By default, all scalar variables with names beginning with I, J, K, L, M, or N are assumed to be default integer variables. Scalar variables with names beginning with any other letter are assumed to be default real variables. For example:

Real Variables	Integer Variables
ALPHA	JCOUNT
BETA	ITEM_1
TOTAL_NUM	NTOTAL

Names beginning with a dollar sign ($) are implicitly INTEGER.

You can override the default data type implied in a name by specifying data type in either an IMPLICIT statement or a type declaration statement.

See Also

- Data Types of Scalar Variables
- Type declaration statements
- IMPLICIT statement

Arrays

An array is a set of scalar elements that have the same type and kind parameters. Any object that is declared with an array specification is an array. Arrays can be declared by using a type declaration statement, or by using a DIMENSION, COMMON, ALLOCATABLE, POINTER, or TARGET statement.

An array can be referenced by element (using subscripts), by section (using a section subscript list), or as a whole. A subscript list (appended to the array name) indicates which array element or array section is being referenced.

A section subscript list consists of subscripts, subscript triplets, or vector subscripts. At least one subscript in the list must be a subscript triplet or vector subscript.

When an array name without any subscripts appears in an intrinsic operation (for example, addition), the operation applies to the whole array (all elements in the array).

An array has the following properties:

- Data type

  An array can have any intrinsic or derived type. The data type of an array (like any other variable) is specified in a type declaration statement or implied by the first letter of its name. All elements of the array have the same type and kind parameters. If a value assigned to an individual array element is not the same as the type of the array, it is converted to the array's type.

- Rank

  The rank of an array is the number of dimensions in the array. An array can have up to seven dimensions. A rank-one array represents a column of data (a vector), a rank-two array represents a table of data arranged in columns and rows (a matrix), a rank-three array represents a table of data on multiple pages (or planes), and so forth.

- Bounds

  Arrays have a lower and upper bound in each dimension. These bounds determine the range of values that can be used as subscripts for the dimension. The value of either bound can be positive, negative, or zero.

  The bounds of a dimension are defined in an array specification.

- Size

  The size of an array is the total number of elements in the array (the product of the array's extents).

  The *extent* is the total number of elements in a particular dimension. It is determined as follows: upper bound - lower bound + 1. If the value of any of an array's extents is zero, the array has a size of zero.

- Shape

  The shape of an array is determined by its rank and extents, and can be represented as a rank-one array (vector) where each element is the extent of the corresponding dimension.

  Two arrays with the same shape are said to be *conformable*. A scalar is conformable to an array of any shape.

The name and rank of an array must be specified when the array is declared. The extent of each dimension can be constant, but does not need to be. The extents can vary during program execution if the array is a dummy argument array, an automatic array, an array pointer, or an allocatable array.

A whole array is referenced by the array name. Individual elements in a named array are referenced by a scalar subscript or list of scalar subscripts (if there is more than one dimension). A section of a named array is referenced by a section subscript.

This section also discusses:

- Whole Arrays
- Array Elements
- Array Sections
- Array Constructors

Examples

The following are examples of valid array declarations:

```
  DIMENSION    A(10, 2, 3)            ! DIMENSION statement
  ALLOCATABLE  B(:, :)                   ! ALLOCATABLE statement
  POINTER      C(:, :, :)                    ! POINTER statement
  REAL, DIMENSION (2, 5) :: D        ! Type declaration with
 DIMENSION attribute
```

Consider the following array declaration:

```
  INTEGER L(2:11,3)
```

The properties of array L are as follows:

Data type:	INTEGER
Rank:	2 (two dimensions)
Bounds:	First dimension: 2 to 11 Second dimension: 1 to 3
Size:	30; the product of the extents: 10 x 3
Shape:	(/10,3/) (or 10 by 3); a vector of the extents 10 and 3

The following example shows other valid ways to declare this array:

```
DIMENSION L(2:11,3)

INTEGER, DIMENSION(2:11,3) :: L

COMMON L(2:11,3)
```

The following example shows references to array elements, array sections, and a whole array:

```
REAL B(10)           ! Declares a rank-one array with 10 elements

INTEGER C(5,8)       ! Declares a rank-two array with 5 elements in

                     !    dimension one and 8 elements in dimension two

...

B(3) = 5.0           ! Reference to an array element

B(2:5) = 1.0         ! Reference to an array section consisting of
                              !    elements: B(2), B(3), B(4), B(5)

...

C(4,8) = I           ! Reference to an array element

C(1:3,3:4) = J       ! Reference to an array section consisting of
                              !   elements:  C(1,3) C(1,4)

                      !             C(2,3) C(2,4)

                      !             C(3,3) C(3,4)/

B = 99                ! Reference to a whole array consisting of

                      !   elements: B(1), B(2), B(3), B(4), B(5),

                      !   B(6), B(7), B(8), B(9), and B(10)
```

Whole Arrays

A *whole array* is a named array; it is either a named constant or a variable. It is referenced by using the array name (without any subscripts).

If a whole array appears in a nonexecutable statement, the statement applies to the entire array. For example:

```
INTEGER, DIMENSION(2:11,3) :: L   ! Specifies the type and

                                  !    dimensions of array L
```

If a whole array appears in an executable statement, the statement applies to all of the elements in the array. For example:

```
L = 10       ! The value 10 is assigned to all the
             !   elements in array L
WRITE *, L   ! Prints all the elements in array L
```

Array Elements

An *array element* is one of the scalar data items that make up an array. A subscript list (appended to the array or array component) determines which element is being referred to. A reference to an array element takes the following form:

array(*subscript-list*)

array	Is the name of the array.
subscript-list	Is a list of one or more subscripts separated by commas. The number of subscripts must equal the rank of the array. Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension.

Description

Each array element inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array element cannot inherit the POINTER attribute.

If an array element is of type character, it can be followed by a substring range in parentheses; for example:

```
ARRAY_D(1,2) (1:3)    ! Elements are substrings of length 3
```

However, by convention, such an object is considered to be a substring rather than an array element.

The following are some valid array element references for an array declared as REAL B(10,20): B(1,3), B(10,10), and B(5,8).

You can use functions and array elements as subscripts. For example:

```
REAL A(3, 3)
REAL B(3, 3), C(89), R
B(2, 2) = 4.5                          ! Assigns the value 4.5 to element B(2, 2)
R = 7.0
C(INT(R)*2 + 1) = 2.0                  ! Element 15 of C = 2.0
A(1,2) = B(INT(C(15)), INT(SQRT(R)))   ! Element A(1,2) = element B(2,2) = 4.5
```

For information on forms for array specifications, see

Declaration Statements for Arrays.

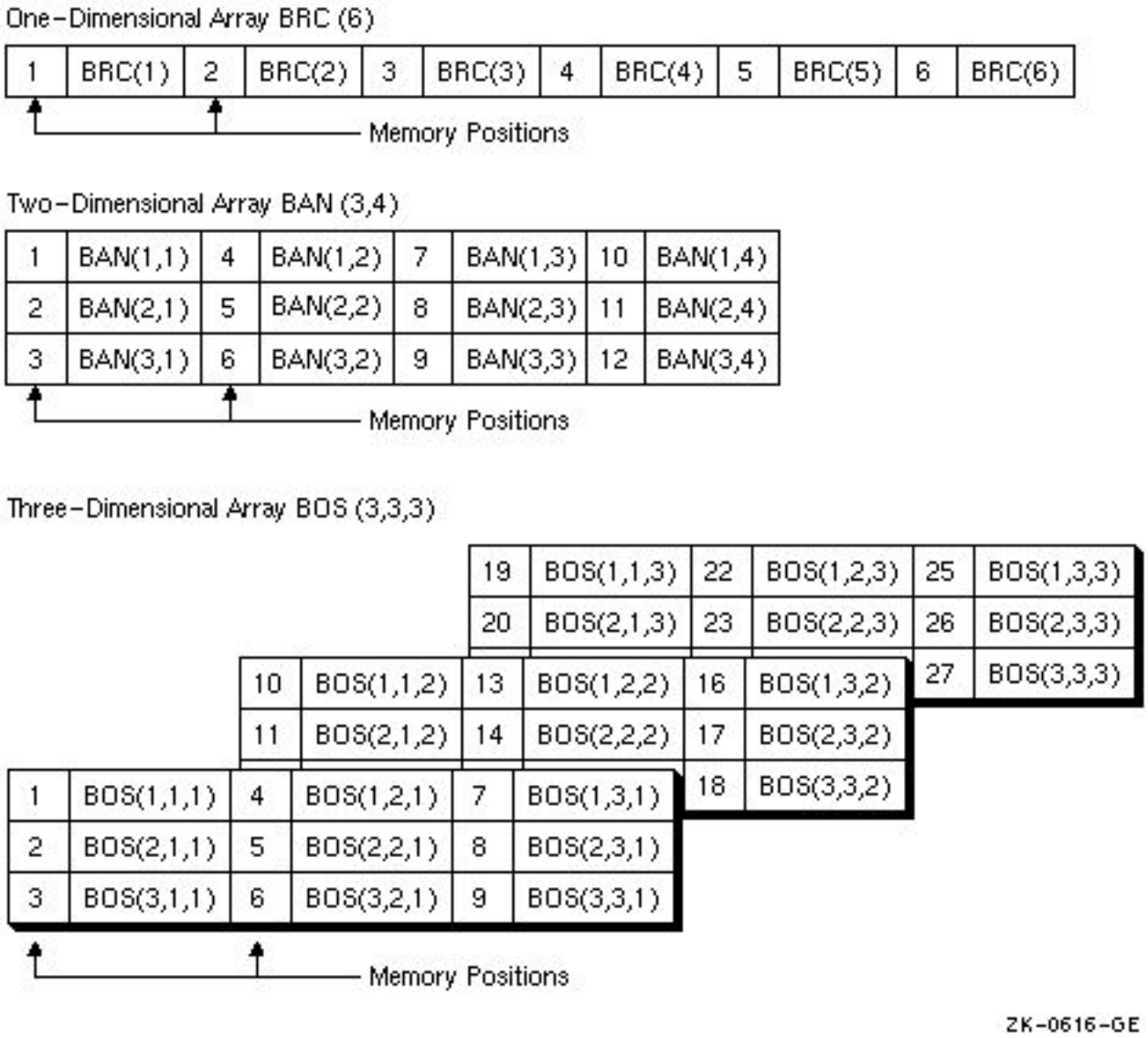
Array Element Order

The elements of an array form a sequence known as array element order. The position of an element in this sequence is its subscript order value.

The elements of an array are stored as a linear sequence of values. A one-dimensional array is stored with its first element in the first storage location and its last element in the last storage location of the sequence. A multidimensional array is stored so that the leftmost subscripts vary most rapidly. This is called the order of subscript progression.

The following figure shows array storage in one, two, and three dimensions:

Figure 27: Array Storage



For example, in two-dimensional array BAN, element BAN(1,2) has a subscript order value of 4; in three-dimensional array BOS, element BOS(1,1,1) has a subscript order value of 1.

In an array section, the subscript order of the elements is their order within the section itself. For example, if an array is declared as B(20), the section B(4:19:4) consists of elements B(4), B(8), B(12), and B(16). The subscript order value of B(4) in the array section is 1; the subscript order value of B(12) in the section is 3.

See Also

- Arrays
- Array association
- Character Constants
- Structure Components
- Storage Association

Array Sections

An *array section* is a portion of an array that is an array itself. It is an array subobject. A section subscript list (appended to the array or array component) determines which portion is being referred to. A reference to an array section takes the following form:

array(*sect-subscript-list*)

array	Is the name of the array.
sect-subscript-list	Is a list of one or more section subscripts (subscripts, subscript triplets, or vector subscripts) indicating a set of elements along a particular dimension. At least one of the items in the section subscript list must be a subscript triplet or vector subscript. A subscript triplet specifies array elements in increasing or decreasing order at a given stride. A vector subscript specifies elements in any order. Each subscript and subscript triplet must be a scalar integer (or other numeric) expression. Each vector subscript must be a rank-one integer expression.

Description

If *no* section subscript list is specified, the rank and shape of the array section is the same as the parent array.

Otherwise, the rank of the array section is the number of vector subscripts and subscript triplets that appear in the list. Its shape is a rank-one array where each element is the number of integer values in the sequence indicated by the corresponding subscript triplet or vector subscript.

If any of these sequences is empty, the array section has a size of zero. The subscript order of the elements of an array section is that of the array object that the array section represents.

Each array section inherits the type, kind type parameter, and certain attributes (INTENT, PARAMETER, and TARGET) of the parent array. An array section cannot inherit the POINTER attribute.

If an array (or array component) is of type character, it can be followed by a substring range in parentheses. Consider the following declaration:

```
CHARACTER(LEN=15) C(10,10)
```

In this case, an array section referenced as C(:,:) (1:3) is an array of shape (10,10), whose elements are substrings of length 3 of the corresponding elements of C.

The following shows valid references to array sections. Note that the syntax (/.../) denotes an array constructor.

```
REAL, DIMENSION(20) :: B

...

PRINT *, B(2:20:5)  ! The section consists of elements

           !     B(2), B(7), B(12), and B(17)

K = (/3, 1, 4/)

B(K) = 0.0      ! Section B(K) is a rank-one array with shape (3) and

       !  size 3. (0.0 is assigned to B(1), B(3), and B(4).)
```

See Also

- Arrays
- Subscript Triplets
- Vector Subscripts
- INTENT
- PARAMETER
- TARGET
- Array constructors
- Character Substrings
- Structure components

Subscript Triplets

A *subscript triplet* is a set of three values representing the lower bound of the array section, the upper bound of the array section, and the increment (stride) between them. It takes the following form:

[*first-bound*] : [*last-bound*] [:*stride*]

first-bound	Is a scalar integer (or other numeric) expression representing the first value in the subscript sequence. If omitted, the declared lower bound of the dimension is used.
last-bound	Is a scalar integer (or other numeric) expression representing the last value in the subscript sequence. If omitted, the declared upper bound of the dimension is used. When indicating sections of an assumed-size array, this subscript *must* be specified.
stride	Is a scalar integer (or other numeric) expression representing the increment between successive subscripts in the sequence. It must have a nonzero value. If it is omitted, it is assumed to be 1.

The stride has the following effects:

- If the stride is positive, the subscript range starts with the first subscript and is incremented by the value of the stride, until the largest value less than or equal to the second subscript is attained.

  For example, if an array has been declared as B(6,3,2), the array section specified as B(2:4,1:2,2) is a rank-two array with shape (3,2) and size 6. It consists of the following six elements:

  ```
  B(2,1,2)    B(2,2,2)

  B(3,1,2)    B(3,2,2)

  B(4,1,2)    B(4,2,2)
  ```

  If the first subscript is greater than the second subscript, the range is empty.

- If the stride is negative, the subscript range starts with the value of the first subscript and is decremented by the absolute value of the stride, until the smallest value greater than or equal to the second subscript is attained.

  For example, if an array has been declared as A(15), the array section specified as A(10:3:-2) is a rank-one array with shape (4) and size 4. It consists of the following four elements:

  ```
  A(10)

  A(8)

  A(6)

  A(4)
  ```

If the second subscript is greater than the first subscript, the range is empty.

If a range specified by the stride is empty, the array section has a size of zero.

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used to select the array elements are within the declared bounds. For example, if an array has been declared as A(15), the array section specified as A(4:16:10) is valid. The section is a rank-one array with shape (2) and size 2. It consists of elements A(4) and A(14).

If the subscript triplet does not specify bounds or stride, but only a colon (:), the entire declared range for the dimension is used.

If you leave out all subscripts, the section defaults to the entire extent in that dimension. For example:

```
REAL A(10)

A(1:5:2) = 3.0   ! Sets elements A(1), A(3), A(5) to 3.0

A(:5:2) = 3.0    ! Same as the previous statement

                 !   because the lower bound defaults to 1

A(2::3) = 3.0    ! Sets elements A(2), A(5), A(8) to 3.0

                 ! The upper bound defaults to 10

A(7:9) = 3.0     ! Sets elements A(7), A(8), A(9) to 3.0

                 ! The stride defaults to 1

A(:) = 3.0       ! Same as A = 3.0; sets all elements of

                 !   A to 3.0
```

See Also

- Array Sections
- Array Sections

Vector Subscripts

A *vector subscript* is a one-dimensional (rank one) array of integer values (within the declared bounds for the dimension) that selects a section of a whole (parent) array. The elements in the section do not have to be in order and the section can contain duplicate values.

For example, A is a rank-two array of shape (4,6). B and C are rank-one arrays of shape (2) and (3), respectively, with the following values:

```
B = (/1,4/)          ! Syntax (/.../) denotes an array constructor

C = (/2,1,1/)        ! This constructor produces a many-one array section
```

Array section A(3,B) consists of elements A(3,1) and A(3,4). Array section A(C,1) consists of elements A(2,1), A(1,1), and A(1,1). Array section A(B,C) consists of the following elements:

```
A(1,2)    A(1,1)    A(1,1)

A(4,2)    A(4,1)    A(4,1)
```

An array section with a vector subscript that has two or more elements with the same value is called a *many-one array section*. For example:

```
REAL A(3, 3), B(4)

INTEGER K(4)

! Vector K has repeated values

K = (/3, 1, 1, 2/)

! Sets all elements of A to 5.0

A = 5.0

B = A(3, K)
```

The array section A(3,K) consists of the elements:

```
A(3, 3)  A(3, 1)  A(3, 1)  A(3, 2)
```

A many-one section must not appear on the left of the equal sign in an assignment statement, or as an input item in a READ statement.

The following assignments to C also show examples of vector subscripts:

```
INTEGER A(2), B(2), C(2)

...

B    = (/1,2/)

C(B) = A(B)

C    = A((/1,2/))
```

An array section with a vector subscript must not be any of the following:

- An internal file
- An actual argument associated with a dummy array that is defined or redefined (if the INTENT attribute is specified, it must be INTENT(IN))
- The target in a pointer assignment statement

If the sequence specified by the vector subscript is empty, the array section has a size of zero.

See Also

- Array Sections
- Array sections
- Array constructors

Array Constructors

An *array constructor* can be used to create and assign values to rank-one arrays (and array constants). An array constructor takes the following form:

(/*ac-value-list*/)

ac-value-list	Is a list of one or more expressions or implied-DO loops. Each *ac-value* must have the same type and kind parameters, and be separated by commas.

An implied-DO loop in an array constructor takes the following form:

(*ac-value-list*, *do-variable* = *expr*1, *expr*2 [,*expr*3])

do-variable	Is the name of a scalar integer variable. Its scope is that of the implied-DO loop.
expr	Is a scalar integer expression. The *expr1* and *expr2* specify a range of values for the loop; *expr3* specifies the stride. The *expr3* must be a nonzero value; if it is omitted, it is assumed to be 1.

Description

The array constructed has the same type as the *ac-value-list* expressions.

If the sequence of values specified by the array constructor is empty (an empty array expression or the implied-DO loop produces no values), the rank-one array has a size of zero.

An *ac-value* is interpreted as follows:

Form of *ac-value*	Result
A scalar expression	Its value is an element of the new array.
An array expression	The values of the elements in the expression (in array element order) are the corresponding sequence of elements in the new array.

Form of *ac-value*	Result
An implied-DO loop	It is expanded to form a list of array elements under control of the DO variable (like a DO construct).

The following shows the three forms of an *ac-value*:

```
C1 = (/4,8,7,6/)                  ! A scalar expression
C2 = (/B(I, 1:5), B(I:J, 7:9)/)   ! An array expression
C3 = (/(I, I=1, 4)/)              ! An implied-DO loop
```

You can also mix these forms, for example:

```
C4 = (/4, A(1:5), (I, I=1, 4), 7/)
```

If every expression in an array constructor is a constant expression, the array constructor is a constant expression.

If the expressions are of type character, Fortran 95/90 requires each expression to have the same character length.

However, Intel Fortran allows the character expressions to be of different character lengths. The length of the resultant character array is the maximum of the lengths of the individual character expressions. For example:

```
print *,len ( (/'a','ab','abc','d'/) )
print *,'++'//(/'a','ab','abc','d'/)//'--'
```

This causes the following to be displayed:

```
           3
 ++a  --++ab --++abc--++d  --
```

If an implied-DO loop is contained within another implied-DO loop (nested), they cannot have the same DO variable (*do-variable*).

To define arrays of more than one dimension, use the RESHAPE intrinsic function.

The following are alternative forms for array constructors:

- Square brackets (instead of parentheses and slashes) to enclose array constructors; for example, the following two array constructors are equivalent:

```
INTEGER C(4)
C = (/4,8,7,6/)
C = [4,8,7,6]
```

- A colon-separated triplet (instead of an implied-DO loop) to specify a range of values and a stride; for example, the following two array constructors are equivalent:

```
INTEGER D(3)
D = (/1:5:2/)              ! Triplet form
D = (/(I, I=1, 5, 2)/)     ! implied-DO loop form
```

Examples

The following example shows an array constructor using an implied-DO loop:

```
INTEGER ARRAY_C(10)
ARRAY_C = (/(I, I=30, 48, 2)/)
```

The values of ARRAY_C are the even numbers 30 through 48.

Implied-DO expressions and values can be mixed in the value list of an array constructor. For example:

```
INTEGER A(10)
A = (/1, 0, (I, I = -1, -6, -1), -7, -8 /)
!Mixed values and implied-DO in value list.
```

This example sets the elements of A to the values, in order, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8.

The following example shows an array constructor of derived type that uses a structure constructor:

```
TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=30) NAME
END TYPE EMPLOYEE
TYPE(EMPLOYEE) CC_4T(4)
CC_4T = (/EMPLOYEE(2732,"JONES"), EMPLOYEE(0217,"LEE"),     &
          EMPLOYEE(1889,"RYAN"), EMPLOYEE(4339,"EMERSON")/)
```

The following example shows how the RESHAPE intrinsic function can be used to create a multidimensional array:

```
E = (/2.3, 4.7, 6.6/)

D = RESHAPE(SOURCE = (/3.5, (/2.0, 1.0/), E/), SHAPE = (/2,3/))
```

D is a rank-two array with shape (2,3) containing the following elements:

```
 3.5    1.0    4.7

 2.0    2.3    6.6
```

The following shows another example:

```
INTEGER B(2,3), C(8)

! Assign values to a (2,3) array.

B = RESHAPE((/1, 2, 3, 4, 5, 6/),(/2,3/))

! Convert B to a vector before assigning values to

! vector C.

C = (/ 0, RESHAPE(B,(/6/)), 7 /)
```

See Also

- Arrays
- DO construct
- Subscript triplets
- Derived types
- Structure constructors
- Array Elements
- Array Assignment Statements
- Declaration Statements for Arrays

Expressions and Assignment Statements

45

This section contains information on expressions and assignment statements.

Expressions

An expression represents either a data reference or a computation, and is formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.

If the value of an expression is of intrinsic type, it has a kind type parameter. (If the value is of intrinsic type CHARACTER, it also has a length parameter.) If the value of an expression is of derived type, it has no kind type parameter.

An operand is a scalar or array. An operator can be either intrinsic or defined. An intrinsic operator is known to the compiler and is always available to any program unit. A defined operator is described explicitly by a user in a function subprogram and is available to each program unit that uses the subprogram.

The simplest form of an expression (a primary) can be any of the following:

- A constant; for example, 4.2
- A subobject of a constant; for example, 'LMNOP' (2:4)
- A variable; for example, VAR_1
- A structure constructor; for example, EMPLOYEE(3472, "JOHN DOE")
- An array constructor; for example, (/12.0,16.0/)
- A function reference; for example, COS(X)
- Another expression in parentheses; for example, (I+5)

Any variable or function reference used as an operand in an expression must be defined at the time the reference is executed. If the operand is a pointer, it must be associated with a target object that is defined. An integer operand must be defined with an integer value rather than a statement label value. All of the characters in a character data object reference must be defined.

When a reference to an array or an array section is made, all of the selected elements must be defined. When a structure is referenced, all of the components must be defined.

In an expression that has intrinsic operators with an array as an operand, the operation is performed on each element of the array. In expressions with more than one array operand, the arrays must be conformable (they must have the same shape). The operation is applied to corresponding elements of the arrays, and the result is an array of the same shape (the same rank and extents) as the operands.

In an expression that has intrinsic operators with a pointer as an operand, the operation is performed on the value of the target associated with the pointer.

For defined operators, operations on arrays and pointers are determined by the procedure defining the operation.

A scalar is conformable with any array. If one operand of an expression is an array and another operand is a scalar, it is as if the value of the scalar were replicated to form an array of the same shape as the array operand. The result is an array of the same shape as the array operand.

The following sections describe numeric, character, relational, and logical expressions; defined operations; a summary of operator precedence; and initialization and specification expressions.

Numeric Expressions

Numeric expressions express numeric computations, and are formed with numeric operands and numeric operators. The evaluation of a numeric operation yields a single numeric value.

The term *numeric* includes logical data, because logical data is treated as integer data when used in a numeric context. The default for .TRUE. is -1; .FALSE. is 0. Note that the default can change if compiler option fpscomp logicals is used.

Numeric operators specify computations to be performed on the values of numeric operands. The result is a scalar numeric value or an array whose elements are scalar numeric values. The following are numeric operators:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus (identity)
-	Subtraction or unary minus (negation)

Unary operators operate on a single operand. *Binary operators* operate on a pair of operands. The plus and minus operators can be unary or binary. When they are unary operators, the plus or minus operators precede a single operand and denote a positive (identity) or negative (negation) value, respectively. The exponentiation, multiplication, and division operators are binary operators.

Valid numeric operations must have results that are defined by the arithmetic used by the processor. For example, raising a negative-valued base to a real power is invalid.

Numeric expressions are evaluated in an order determined by a precedence associated with each operator, as follows (see also Summary of Operator Precedence):

Operator	Precedence
**	Highest
* and /	.
Unary + and -	.
Binary + and -	Lowest

Operators with equal precedence are evaluated in left-to-right order. However, exponentiation is evaluated from right to left. For example, A**B**C is evaluated as A**(B**C). B**C is evaluated first, then A is raised to the resulting power.

Normally, two operators cannot appear together. However, Intel© Fortran allows two consecutive operators if the second operator is a plus or minus.

Examples

In the following example, the exponentiation operator is evaluated first because it takes precedence over the multiplication operator:

`A**B*C` is evaluated as `(A**B)*C`

Ordinarily, the exponentiation operator would be evaluated first in the following example. However, because Intel Fortran allows the combination of the exponentiation and minus operators, the exponentiation operator is not evaluated until the minus operator is evaluated:

`A**-B*C` is evaluated as `A**(-(B*C))`

Note that the multiplication operator is evaluated first, since it takes precedence over the minus operator.

When consecutive operators are used with constants, the unary plus or minus before the constant is treated the same as any other operator. This can produce unexpected results. In the following example, the multiplication operator is evaluated first, since it takes precedence over the minus operator:

`X/-15.0*Y` is evaluated as `X/-(15.0*Y)`

Using Parentheses in Numeric Expressions

You can use parentheses to force a particular order of evaluation. When part of an expression is enclosed in parentheses, that part is evaluated first. The resulting value is used in the evaluation of the remainder of the expression.

In the following examples, the numbers below the operators indicate a possible order of evaluation. Alternative evaluation orders are possible in the first three examples because they contain operators of equal precedence that are not enclosed in parentheses. In these cases, the compiler is free to evaluate operators of equal precedence in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation.

```
4 + 3 * 2 - 6/2   = 7
  ^   ^   ^  ^

  2   1   4  3

(4 + 3) * 2 - 6/2   = 11
   ^    ^   ^  ^

   1    2   4  3

(4 + 3 * 2 - 6)/2   = 2
   ^   ^   ^   ^

   2   1   3   4

((4 + 3) * 2 - 6)/2   = 4
    ^    ^   ^   ^

    1    2   3   4
```

Expressions within parentheses are evaluated according to the normal order of precedence. In expressions containing nested parentheses, the innermost parentheses are evaluated first.

Nonessential parentheses do not affect expression evaluation, as shown in the following example:

```
4 + (3 * 2) - (6/2)
```

However, using parentheses to specify the evaluation order is often important in high-accuracy numerical computations. In such computations, evaluation orders that are algebraically equivalent may not be computationally equivalent when processed by a computer (because of the way intermediate results are rounded off).

Parentheses can be used in argument lists to force a given argument to be treated as an expression, rather than as the address of a memory item.

Data Type of Numeric Expressions

If every operand in a numeric expression is of the same data type, the result is also of that type.

If operands of different data types are combined in an expression, the evaluation of that expression and the data type of the resulting value depend on the ranking associated with each data type. The following table shows the ranking assigned to each data type:

Data Type	Ranking
LOGICAL(1) and BYTE	Lowest
LOGICAL(2)	.
LOGICAL(4)	.
LOGICAL(8)	.
INTEGER(1)	.
INTEGER(2)	.
INTEGER(4)	.
INTEGER(8)	.
REAL(4)	.
REAL(8)[1]	.
REAL(16)	.
COMPLEX(4)	.
COMPLEX(8)[2]	.
COMPLEX(16)	Highest

Data Type	Ranking
[1] DOUBLE PRECISION	
[2] DOUBLE COMPLEX	

The data type of the value produced by an operation on two numeric operands of different data types is the data type of the highest-ranking operand in the operation. For example, the value resulting from an operation on an integer and a real operand is of real type. However, an operation involving a COMPLEX(4) or COMPLEX(8) data type and a DOUBLE PRECISION data type produces a COMPLEX(8) result.

The data type of an expression is the data type of the result of the last operation in that expression, and is determined according to the following conventions:

- Integer operations: Integer operations are performed only on integer operands. (Logical entities used in a numeric context are treated as integers.) In integer arithmetic, any fraction resulting from division is truncated, not rounded. For example, the result of `9/10` is 0, not 1.
- Real operations: Real operations are performed only on real operands or combinations of real, integer, and logical operands. Any integer operands present are converted to real data type by giving each a fractional part equal to zero. The expression is then evaluated using real arithmetic. However, in the statement `Y = (I /J)*X`, an integer division operation is performed on I and J, and a real multiplication is performed on that result and X.

  If one operand is a higher-precision real (REAL(8) or REAL(16)) type, the other operand is converted to that higher-precision real type before the expression is evaluated.

  When a single-precision real operand is converted to a double-precision real operand, low-order binary digits are set to zero. This conversion does not increase accuracy; conversion of a decimal number does not produce a succession of decimal zeros. For example, a REAL variable having the value `0.3333333` is converted to approximately `0.33333331346511840`. It is not converted to either `0.3333333000000000D0` or `0.3333333333333333D0`.
- Complex operations: In operations that contain any complex operands, integer operands are converted to real type, as previously described. The resulting single-precision or double-precision operand is designated as the real part of a complex number and the imaginary part is assigned a value of zero. The expression is then evaluated using complex arithmetic and the resulting value is of complex type. Operations involving a COMPLEX(4) or COMPLEX(8) operand and a DOUBLE PRECISION operand are performed as COMPLEX(8) operations; the DOUBLE PRECISION operand is not rounded.

These rules also generally apply to numeric operations in which one of the operands is a constant. However, if a real or complex constant is used in a higher-precision expression, additional precision will be retained for the constant. The effect is as if a DOUBLE PRECISION (REAL(8)) or REAL(16) representation of the constant were given. For example, the expression `1.0D0 + 0.3333333` is treated as if it is `1.0D0 + dble(0.3333333)`.

Character Expressions

A character expression consists of a character operator (//) that concatenates two operands of type character. The evaluation of a character expression produces a single value of that type.

The result of a character expression is a character string whose value is the value of the left character operand concatenated to the value of the right operand. The length of a character expression is the sum of the lengths of the values of the operands. For example, the value of the character expression `'AB'//'CDE'` is `'ABCDE'`, which has a length of five.

Parentheses do not affect the evaluation of a character expression; for example, the following character expressions are equivalent:

```
('ABC'//'DE')//'F'

'ABC'//('DE'//'F')

'ABC'//'DE'//'F'
```

Each of these expressions has the value `' ABCDEF'`.

If a character operand in a character expression contains blanks, the blanks are included in the value of the character expression. For example, `'ABC '//'D E'//'F '` has a value of `'ABC D EF '`.

Relational Expressions

A *relational expression* consists of two or more expressions whose values are compared to determine whether the relationship stated by the relational operator is satisfied. The following are relational operators:

Operator	Relationship
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or = =	Equal to
.NE. or /=	Not equal to

Operator	Relationship
.GT. or >	Greater than
.GE. or >=	Greater than or equal to

The result of the relational expression is .TRUE. if the relation specified by the operator is satisfied; the result is .FALSE. if the relation specified by the operator is not satisfied.

Relational operators are of equal precedence. Numeric operators and the character operator // have a higher precedence than relational operators.

In a numeric relational expression, the operands are numeric expressions. Consider the following example:

```
APPLE+PEACH > PEAR+ORANGE
```

This expression states that the sum of APPLE and PEACH is greater than the sum of PEAR and ORANGE. If this relationship is valid, the value of the expression is .TRUE.; if not, the value is .FALSE.

Operands of type complex can only be compared using the equal operator (= = or .EQ.) or the not equal operator (/= or .NE.). Complex entities are equal if their corresponding real and imaginary parts are both equal.

In a character relational expression, the operands are character expressions. In character relational expressions, less than (< or .LT.) means the character value precedes in the ASCII collating sequence, and greater than (> or .GT.) means the character value follows in the ASCII collating sequence. For example:

```
'AB'//'ZZZ' .LT. 'CCCCC'
```

This expression states that `'ABZZZ'` is less than `'CCCCC'`. In this case, the relation specified by the operator is satisfied, so the result is .TRUE..

Character operands are compared one character at a time, in order, starting with the first character of each operand. If the two character operands are not the same length, the shorter one is padded on the right with blanks until the lengths are equal; for example:

```
'ABC' .EQ. 'ABC  '

'AB' .LT. 'C'
```

The first relational expression has the value .TRUE. even though the lengths of the expressions are not equal, and the second has the value .TRUE. even though `'AB'` is longer than `'C'`.

A relational expression can compare two numeric expressions of different data types. In this case, the value of the expression with the lower-ranking data type is converted to the higher-ranking data type before the comparison is made.

See Also

- Expressions
- Data Type of Numeric Expressions

Logical Expressions

A logical expression consists of one or more logical operators and logical, numeric, or relational operands. The following are logical operators:

Operator	Example	Meaning
.AND.	`A .AND. B`	Logical conjunction: the expression is true if both A and B are true.
.OR.	`A .OR. B`	Logical disjunction (inclusive OR): the expression is true if either A, B, or both, are true.
.NEQV.	`A .NEQV. B`	Logical inequivalence (exclusive OR): the expression is true if either A or B is true, but false if both are true.
.XOR.	`A .XOR. B`	Same as .NEQV.
.EQV.	`A .EQV. B`	Logical equivalence: the expression is true if both A and B are true, or both are false.
.NOT.[1]	`.NOT. A`	Logical negation: the expression is true if A is false and false if A is true.

[1] .NOT. is a unary operator.

Periods cannot appear consecutively except when the second operator is .NOT. For example, the following logical expression is valid:

```
A+B/(A-1) .AND. .NOT. D+B/(D-1)
```

Data Types Resulting from Logical Operations

Logical operations on logical operands produce single logical values (.TRUE. or .FALSE.) of logical type.

Logical operations on integers produce single values of integer type. The operation is carried out bit-by-bit on corresponding bits of the internal (binary) representation of the integer operands.

Logical operations on a combination of integer and logical values also produce single values of integer type. The operation first converts logical values to integers, then operates as it does with integers.

Logical operations cannot be performed on other data types.

Evaluation of Logical Expressions

Logical expressions are evaluated according to the precedence of their operators. Consider the following expression:

```
A*B+C*ABC == X*Y+DM/ZZ .AND. .NOT. K*B > TT
```

This expression is evaluated in the following sequence:

```
(((A*B)+(C*ABC)) == ((X*Y)+(DM/ZZ))) .AND. (.NOT. ((K*B) > TT))
```

As with numeric expressions, you can use parentheses to alter the sequence of evaluation.

When operators have equal precedence, the compiler can evaluate them in any order, as long as the result is the same as the result gained by the algebraic left-to-right order of evaluation (except for exponentiation, which is evaluated from right to left).

You should not write logical expressions whose results might depend on the evaluation order of subexpressions. The compiler is free to evaluate subexpressions in any order. In the following example, either (A(I)+1.0) or B(I)*2.0 could be evaluated first:

```
(A(I)+1.0) .GT. B(I)*2.0
```

Some subexpressions might not be evaluated if the compiler can determine the result by testing other subexpressions in the logical expression. Consider the following expression:

```
A .AND. (F(X,Y) .GT. 2.0) .AND. B
```

If the compiler evaluates A first, and A is false, the compiler might determine that the expression is false and might not call the subprogram F(X,Y).

See Also

- Expressions

- Summary of Operator Precedence

Defined Operations

When operators are defined for functions, the functions can then be referenced as defined operations.

The operators are defined by using a generic interface block specifying OPERATOR, followed by the defined operator (in parentheses).

A defined operation is not an intrinsic operation. However, you can use a defined operation to extend the meaning of an intrinsic operator.

For defined unary operations, the function must contain one argument. For defined binary operations, the function must contain two arguments.

Interpretation of the operation is provided by the function that defines the operation.

A Fortran 95/90 defined operator can contain up to 31 letters, and is enclosed in periods (.). Its name cannot be the same name as any of the following:

- The intrinsic operators (.NOT., .AND., .OR., .XOR., .EQV., .NEQV., .EQ., .NE., .GT., .GE., .LT., and .LE.)
- The logical literal constants (.TRUE. or .FALSE.)

An intrinsic operator can be followed by a defined unary operator.

The result of a defined operation can have any type. The type of the result (and its value) must be specified by the defining function.

Examples

The following examples show expressions containing defined operators:

```
.COMPLEMENT. A

X .PLUS. Y .PLUS. Z

M * .MINUS. N
```

See Also

- Expressions
- Defining Generic Operators
- Summary of Operator Precedence

Summary of Operator Precedence

The following table shows the precedence of all intrinsic and defined operators:

Precedence of Expression Operators

Category	Operator	Precedence
	Defined Unary Operators	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	Unary + or -	.
Numeric	Binary + or -	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., = =, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.XOR., .EQV., .NEQV.	.
	Defined Binary Operators	Lowest

Initialization and Specification Expressions

A constant expression contains intrinsic operations and parts that are all constants. An initialization expression is a constant expression that is evaluated when a program is compiled. A specification expression is a scalar, integer expression that is restricted to declarations of array bounds and character lengths.

Initialization and specification expressions can appear in specification statements, with some restrictions.

Initialization Expressions

An initialization expression must evaluate at compile time to a constant. It is used to specify an initial value for an entity.

In an initialization expression, each operation is intrinsic and each operand is one of the following:

- A constant or subobject of a constant
- An array constructor where each element and the bounds and strides of each implied-DO, are expressions whose primaries are initialization expressions
- A structure constructor whose components are initialization expressions
- An elemental intrinsic function reference of type integer or character, whose arguments are initialization expressions of type integer or character
- A reference to one of the following inquiry functions:

BIT_SIZE	MINEXPONENT
DIGITS	PRECISION
EPSILON	RADIX
HUGE	RANGE
ILEN	SHAPE
KIND	SIZE
LBOUND	TINY
LEN	UBOUND
MAXEXPONENT	

  Each function argument must be one of the following:

  - An initialization expression
  - A variable whose kind type parameter and bounds are not assumed or defined by an ALLOCATE statement, pointer assignment, or an expression that is not an initialization expression

- A reference to one of the following transformational functions (each argument must be an initialization expression):

REPEAT	SELECTED_REAL_KIND
RESHAPE	TRANSFER
SELECTED_CHAR_KIND	TRIM

SELECTED_INT_KIND

- A reference to the transformational function NULL
- An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are initialization expressions
- Another initialization expression enclosed in parentheses

Each subscript, section subscript, and substring starting and ending point must be an initialization expression.

In an initialization expression, the exponential operator (**) must have a power of type integer.

If an initialization expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement (or to the left of the inquiry function in the same statement).

Examples

Table 567: Valid Initialization (Constant) Expressions

`-1 + 3`	
`SIZE(B)`	! B is a named constant
`7_2`	
`INT(J, 4)`	! J is a named constant
`SELECTED_INT_KIND (2)`	

Table 568: Invalid Initialization (Constant) Expressions

`SUM(A)`	Not an allowed function.
`A/4.1 - K**1.2`	Exponential does not have integer power (A and K are named constants).
`HUGE(4.0)`	Argument is not an integer.

See Also

- Initialization and Specification Expressions
- Array constructors

- Structure constructors
- Intrinsic procedures

Specification Expressions

A specification expression is a restricted expression that is of type integer and has a scalar value. This type of expression appears only in the declaration of array bounds and character lengths.

In a restricted expression, each operation is intrinsic and each operand is one of the following:

- A constant or subobject of a constant
- A variable that is one of the following:
  - A dummy argument that does not have the OPTIONAL or INTENT (OUT) attribute (or the subobject of such a variable)
  - In a common block (or the subobject of such a variable)
  - Made accessible by use or host association (or the subobject of such a variable)
- A structure constructor whose components are restricted expressions
- An implied-DO variable within an array constructor, where the bounds and strides of the corresponding implied-DO are restricted expressions
- A reference to one of the following inquiry functions:

BIT_SIZE	MINEXPONENT
DIGITS	PRECISION
EPSILON	RADIX
HUGE	RANGE
ILEN	SHAPE
KIND	SIZE
LBOUND	SIZEOF
LEN	TINY
MAXEXPONENT	UBOUND

Each function argument must be one of the following:

- A restricted expression

    - A variable whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, are not defined by an expression that is not a restricted expression, or are not definable by an ALLOCATE or pointer assignment statement.

- A reference to any other intrinsic function where each argument is a restricted expression.
- A reference to a specification function where each argument is a restricted expression
- An array constructor where each element and the bounds and strides of each implied-DO, are expressions whose primaries are restricted expressions
- Another restricted expression enclosed in parentheses

Each subscript, section subscript, and substring starting and ending point must be a restricted expression.

Specification functions can be used in specification expressions to indicate the attributes of data objects. A specification function is a pure function. It cannot have a dummy procedure argument or be any of the following:

- An intrinsic function
- An internal function
- A statement function
- Defined as RECURSIVE

A variable in a specification expression must have its type and type parameters (if any) specified in one of the following ways:

- By a previous declaration in the same scoping unit
- By the implicit typing rules currently in effect for the scoping unit
- By host or use association

If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm the implied type and type parameters.

If a specification expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement (or to the left of the inquiry function in the same statement).

In a specification expression, the number of arguments for a function reference is limited to 255.

Examples

The following shows valid specification expressions:

```
MAX(I) + J          ! I and J are scalar integer variables

UBOUND(ARRAY_B,20)  ! ARRAY_B is an assumed-shape dummy array
```

See Also

- Initialization and Specification Expressions
- Array constructors
- Structure constructors
- Intrinsic procedures
- Implicit typing rules
- Use and host association
- PURE procedures

Assignment Statements

An assignment statement causes variables to be defined or redefined. This section describes the following kinds of assignment statements: intrinsic, defined, pointer, masked array (WHERE), and element array (FORALL).

The ASSIGN statement assigns a label to an integer variable. It is discussed elsewhere.

Intrinsic Assignments

Intrinsic assignment is used to assign a value to a nonpointer variable. In the case of pointers, intrinsic assignment is used to assign a value to the target associated with the pointer variable. The value assigned to the variable (or target) is determined by evaluation of the expression to the right of the equal sign.

An intrinsic assignment statement takes the following form:

variable = *expression*

variable	Is the name of a scalar or array of intrinsic or derived type (with no defined assignment). The array cannot be an assumed-size array, and neither the scalar nor the array can be declared with the PARAMETER or INTENT(IN) attribute.
expression	Is of intrinsic type or the same derived type as *variable*. Its shape must conform with *variable*. If necessary, it is converted to the same type and kind as *variable*.

Description

Before a value is assigned to the variable, the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement.

NOTE. When the run-time system assigns a value to a scalar integer or character variable and the variable is shorter than the value being assigned, the assigned value may be truncated and significant bits (or characters) lost. This truncation can occur without warning, and can cause the run-time system to pass incorrect information back to the program.

If the variable is a pointer, it must be associated with a definable target. The shape of the target and expression must conform and their type and kind parameters must match.

The following sections discuss numeric, logical, character, derived- type, and array intrinsic assignment.

Numeric Assignment Statements

For numeric assignment statements, the variable and expression must be numeric type.

The expression must yield a value that conforms to the range requirements of the variable. For example, a real expression that produces a value greater than 32767 is invalid if the entity on the left of the equal sign is an INTEGER(2) variable.

Significance can be lost if an INTEGER(4) value, which can exactly represent values of approximately the range -2*10**9 to +2*10**9, is converted to REAL(4) (including the real part of a complex constant), which is accurate to only about seven digits.

If the variable has the same data type as that of the expression on the right, the statement assigns the value directly. If the data types are different, the value of the expression is converted to the data type of the variable before it is assigned.

The following table summarizes the data conversion rules for numeric assignment statements.

Table 570: Conversion Rules for Numeric Assignment Statements

Scalar Memory Reference (V)	Expression (E)	
	Integer, Logical or Real	Complex
Integer or Logical	V=INT(E)	V=INT(REAL(E))

Scalar Memory Reference (V)	Expression (E)	
	Integer, Logical or Real	Complex
		Imaginary part of E is not used.
REAL (KIND=4)	V=REAL(E)	V=REAL(REAL(E)) Imaginary part of E is not used.
REAL (KIND=8)	V=DBLE(E)	V=DBLE(REAL(E)) Imaginary part of E is not used.
REAL (KIND=16)	V=QEXT(E)	V=QEXT(REAL(E)) Imaginary part of E is not used.
COMPLEX (KIND=4)	V=CMPLX(REAL(E), 0.0)	V=CMPLX(REAL(REAL(E)), REAL(AIMAG(E)))
COMPLEX (KIND=8)	V=CMPLX(DBLE(E), 0.0)	V=CMPLX(DBLE(REAL(E)), DBLE(AIMAG(E)))
COMPLEX (KIND=16)	V=CMPLX(QEXT(E), 0.0)	V=CMPLX(QEXT(REAL(E)), QEXT(AIMAG(E)))

Examples

Table 571: Valid Numeric Assignment Statements

```
BETA = -1./(2.*X)+A*A /(4.*(X*X))

PI = 3.14159

SUM = SUM + 1.

ARRAY_A = ARRAY_B + ARRAY_C + SCALAR_I    ! Valid if all arrays conform in shape
```

Table 572: Invalid Numeric Assignment Statements

3.14 = A - B	Entity on the left must be a variable.
ICOUNT = A//B(3:7)	Implicitly typed data types do not match.
SCALAR_I = ARRAY_A(:)	Shapes do not match.

See Also

- Intrinsic Assignments
- INT
- REAL
- DBLE
- QEXT
- CMPLX
- AIMAG

Logical Assignment Statements

For logical assignment statements, the variable must be of logical type and the expression can be of logical or numeric type.

If necessary, the expression is converted to the same type and kind as the variable.

Examples

The following examples demonstrate valid logical assignment statements:

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A.GT.B .AND. A.GT.C .AND. A.GT.D

LOGICAL_VAR = 123      ! Moves binary value of 123 to LOGICAL_VAR
```

Character Assignment Statements

For character assignment statements, the variable and expression must be of character type and have the same kind parameter.

The variable and expression can have different lengths. If the length of the expression is greater than the length of the variable, the character expression is truncated on the right. If the length of the expression is less than the length of the variable, the character expression is filled on the right with blank characters.

If you assign a value to a character substring, you do not affect character positions in any part of the character scalar variable not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged. If the character position is undefined, it remains undefined.

Examples

Table 573: Valid Character Assignment Statements. (All variables are of type character.)

```
FILE = 'PROG2'

REVOL(1) = 'MAR'//'CIA'

LOCA(3:8) = 'PLANT5'

TEXT(I,J+1)(2:N-1) = NAME/ /X
```

Table 574: Invalid Character Assignment Statements

`'ABC' = CHARS`	Left element must be a character variable, array element, or substring reference.
`CHARS = 25`	Expression does not have a character data type.
`STRING=5HBEGIN`	Expression does not have a character data type. (Hollerith constants are numeric, not character.)

Derived-Type Assignment Statements

In derived-type assignment statements, the variable and expression must be of the same derived type. There must be no accessible interface block with defined assignment for objects of this derived type.

The derived-type assignment is performed as if each component of the expression is assigned to the corresponding component of the variable. Pointer assignment is performed for pointer components, and intrinsic assignment is performed for nonpointer components.

Examples

The following example shows derived-type assignment:

```
TYPE DATE
  LOGICAL(1) DAY, MONTH
  INTEGER(2) YEAR
END TYPE DATE
TYPE(DATE) TODAY, THIS_WEEK(7)
TYPE APPOINTMENT
...
  TYPE(DATE) APP_DATE
END TYPE
TYPE(APPOINTMENT) MEETING
DO I = 1,7
  CALL GET_DATE(TODAY)
  THIS_WEEK(I) = TODAY
END DO
MEETING%APP_DATE = TODAY
```

See Also

- Intrinsic Assignments
- Derived types
- Pointer assignments

Array Assignment Statements

Array assignment is permitted when the array expression on the right has the same shape as the array variable on the left, or the expression on the right is a scalar.

If the expression is a scalar, and the variable is an array, the scalar value is assigned to every element of the array.

If the expression is an array, the variable must also be an array. The array element values of the expression are assigned (element by element) to corresponding elements of the array variable.

A *many-one array section* is a vector-valued subscript that has two or more elements with the same value. In intrinsic assignment, the variable cannot be a many-one array section because the result of the assignment is undefined.

Examples

In the following example, X and Y are arrays of the same shape:

```
X = Y
```

The corresponding elements of Y are assigned to those of X element by element; the first element of Y is assigned to the first element of X, and so forth. The processor can perform the element-by-element assignment in any order.

The following example shows a scalar assigned to an array:

```
B(C+1:N, C) = 0
```

This sets the elements B (C+1,C), B (C+2,C),...B (N,C) to zero.

The following example causes the values of the elements of array A to be reversed:

```
REAL A(20)

...

A(1:20) = A(20:1:-1)
```

See Also

- Intrinsic Assignments
- Arrays
- Array constructors
- WHERE
- FORALL

Defined Assignments

Defined assignment specifies an assignment operation. It is defined by a subroutine subprogram containing a generic interface block with the specifier ASSIGNMENT(=). The subroutine is specified by a SUBROUTINE or ENTRY statement that has two nonoptional dummy arguments.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

The dummy arguments represent the variable and expression, in that order. The rank (and shape, if either or both are arrays), type, and kind parameters of the variable and expression in the assignment statement must match those of the corresponding dummy arguments.

The dummy arguments must not both be numeric, or of type logical or character with the same kind parameter.

If the variable in an elemental assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of the variable and expression. If the expression is scalar, it is treated as if it were an array of the same shape as the variable with every element of the array equal to the scalar value of the expression.

See Also

- Assignment Statements
- Subroutines
- Derived data types
- Defining Generic Assignment
- Numeric Expressions
- Character Expressions

Pointer Assignments

In ordinary assignment involving pointers, the pointer is an alias for its target. In pointer assignment, the pointer is associated with a target. If the target is undefined or disassociated, the pointer acquires the same status as the target. The pointer assignment statement has the following form:

pointer-object => *target*

pointer-object	Is a variable name or structure component declared with the POINTER attribute.
target	Is a variable or expression. Its type and kind parameters, and rank must be the same as *pointer-object*. It cannot be an array section with a vector subscript.

Description

If the target is a variable, it must have the POINTER or TARGET attribute, or be a subobject whose parent object has the TARGET attribute.

If the target is an expression, the result must be a pointer.

If the target is not a pointer (it has the TARGET attribute), the pointer object is associated with the target.

If the target is a pointer (it has the POINTER attribute), its status determines the status of the pointer object, as follows:

- If the pointer is associated, the pointer object is associated with the same object as the target
- If the pointer is disassociated, the pointer object becomes disassociated
- If the pointer is undefined, the pointer object becomes undefined

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined.

When pointer assignment occurs, any previous association between the pointer object and a target is terminated.

Pointers can also be assigned for a pointer structure component by execution of a derived-type intrinsic assignment statement or a defined assignment statement.

Pointers can also become associated by using the ALLOCATE statement to allocate the pointer.

Pointers can become disassociated by deallocation, nullification of the pointer (using the DEALLOCATE or NULLIFY statements), or by reference to the NULL intrinsic function.

Examples

The following are examples of pointer assignments:

```
HOUR => MINUTES(1:60)            ! target is an array
M_YEAR => MY_CAR%YEAR          ! target is a structure component
NEW_ROW%RIGHT => CURRENT_ROW     ! pointer object is a structure component
PTR => M                              ! target is a variable
POINTER_C => NULL ()           ! reference to NULL intrinsic
```

The following example shows a target as a pointer:

```
INTEGER, POINTER :: P, N
INTEGER, TARGET :: M
INTEGER S
M = 14
N => M                          ! N is associated with M
P => N                           ! P is associated with M through N
S = P + 5
```

The value assigned to S is 19 (14 + 5).

You can use the intrinsic function ASSOCIATED to find out if a pointer is associated with a target or if two pointers are associated with the same target. For example:

```
REAL C (:), D(:), E(5)

POINTER C, D

TARGET E

LOGICAL STATUS

! Pointer assignment.

C => E

! Pointer assignment.

D => E

! Returns TRUE; C is associated.

STATUS = ASSOCIATED (C)

! Returns TRUE; C is associated with E.

STATUS = ASSOCIATED (C, E)

! Returns TRUE; C and D are associated with the

! same target.

STATUS = ASSOCIATED (C, D)
```

See Also

- Assignment Statements
- Arrays
- ALLOCATE
- DEALLOCATE
- NULLIFY
- NULL
- POINTER
- TARGET
- Defined assignments
- Intrinsic Assignments

WHERE Statement and Construct Overview

You can perform an array operation on selected elements by using masked array assignment. For more information, see WHERE.

See Also

- Assignment Statements
- FORALL

FORALL Statement and Construct Overview

The FORALL statement and construct is a generalization of the Fortran 95/90 masked array assignment. It allows more general array shapes to be assigned, especially in construct form. For more information, see FORALL.

See Also

- Assignment Statements
- WHERE

Specification Statements

46

A *specification statement* is a nonexecutable statement that declares the attributes of data objects. In Fortran 95/90, many of the attributes that can be defined in specification statements can also be optionally specified in type declaration statements.

The following are specification statements:

- Type declaration statement

  Explicitly specifies the properties (for example: data type, rank, and extent) of data objects.
- ALLOCATABLE attribute and statement

  Specifies a list of array names that are allocatable (have a deferred-shape).
- ASYNCHRONOUS attribute and statement

  Specifies that a variable can be used for asynchronous input and output.
- AUTOMATIC and STATIC attributes and statements

  Control the storage allocation of variables in subprograms.
- BIND attribute and statement

  Specifies that an object is interoperable with C and has external linkage.
- COMMON statement

  Defines one or more contiguous areas, or blocks, of physical storage (called common blocks).
- DATA statement

  Assigns initial values to variables before program execution.
- DIMENSION attribute and statement

  Specifies that an object is an array, and defines the shape of the array.
- EQUIVALENCE statement

  Specifies that a storage area is shared by two or more objects in a program unit.
- EXTERNAL attribute and statement

  Allows external (user-supplied) procedures to be used as arguments to other subprograms.
- IMPLICIT statement

  Overrides the implicit data type of names.
- INTENT attribute and statement

  Specifies the intended use of a dummy argument.

- INTRINSIC attribute and statement

  Allows intrinsic procedures to be used as arguments to subprograms.
- NAMELIST statement

  Associates a name with a list of variables. This group name can be referenced in some input/output operations.
- OPTIONAL attribute and statement

  Allows a procedure reference to omit arguments.
- PARAMETER attribute and statement

  Defines a named constant.
- POINTER attribute and statement

  Specifies that an object is a pointer.
- PRIVATE and PUBLIC and attributes and statements

  Declare the accessibility of entities in a module.
- PROTECTED attribute and statement

  Specifies limitations on the use of module entities.
- SAVE attribute and statement

  Causes the definition and status of objects to be retained after the subprogram in which they are declared completes execution.
- TARGET attribute and statement

  Specifies a pointer target.
- VALUE attribute and statement

  Specifies a type of argument association for a dummy argument.
- VOLATILE attribute and statement

  Prevents optimizations from being performed on specified objects.

Type Declaration Statements

A type declaration statement explicitly specifies the properties of data objects or functions. For more information, see Type Declarations.

See Also

- Specification Statements

- Declaration Statements for Noncharacter Types
- Declaration Statements for Character Types
- Declaration Statements for Derived Types
- Declaration Statements for Arrays
- Declaration Statements for Noncharacter Types
- Declaration Statements for Character Types
- Declaration Statements for Derived Types
- Declaration Statements for Arrays
- Derived data types
- DATA
- Initialization expressions
- Intrinsic Data Types
- Implicit Typing Rules
- Specification of Data Type

Declaration Statements for Noncharacter Types

The following table shows the data types that can appear in noncharacter type declaration statements.

Noncharacter Data Types
BYTE[1]
LOGICAL[2]
LOGICAL([KIND=]1) (or LOGICAL*1)
LOGICAL([KIND=]2) (or LOGICAL*2)
LOGICAL([KIND=]4) (or LOGICAL*4)
LOGICAL([KIND=]8) (or LOGICAL*8)
INTEGER[3]
INTEGER([KIND=]1) (or INTEGER*1)
INTEGER([KIND=]2) (or INTEGER*2)
INTEGER([KIND=]4) (or INTEGER*4)

Noncharacter Data Types
INTEGER([KIND=]8) (or INTEGER*8)
REAL[4]
REAL([KIND=]4) (or REAL*4)
DOUBLE PRECISION (REAL([KIND=]8) or REAL*8)
REAL([KIND=]16) (or REAL*16)
COMPLEX[5]
COMPLEX([KIND=]4) (or COMPLEX*8)
DOUBLE COMPLEX (COMPLEX([KIND=]8) or COMPLEX*16)
COMPLEX([KIND=]16) (or COMPLEX*32)

[1] Same as INTEGER(1).

[2] This is treated as default logical.

[3] This is treated as default integer.

[4] This is treated as default real.

[5] This is treated as default complex.

In noncharacter type declaration statements, you can optionally specify the name of the data object or function as v*n, where *n* is the length (in bytes) of *v*. The length specified overrides the length implied by the data type.

The value for *n* must be a valid length for the type of *v*. The type specifiers BYTE, DOUBLE PRECISION, and DOUBLE COMPLEX have one valid length, so the *n* specifier is invalid for them.

For an array specification, the *n* must be placed immediately following the array name; for example, in an INTEGER declaration statement, IVEC*2(10) is an INTEGER(2) array of 10 elements.

Note that certain compiler options can affect the defaults for numeric and logical data types.

Examples

In a noncharacter type declaration statement, a subsequent kind parameter overrides any initial kind parameter. For example, consider the following statements:

```
INTEGER(KIND=2) I, J, K, M12*4, Q, IVEC*4(10)

REAL(KIND=8) WX1, WXZ, WX3*4, WX5, WX6*4

REAL(KIND=8) PI/3.14159E0/, E/2.72E0/, QARRAY(10)/5*0.0,5*1.0/
```

In the first statement, M12*4 and IVEC*4 override the KIND=2 specification. In the second statement, WX3*4 and WX6*4 override the KIND=8 specification. In the third statement, QARRAY is initialized with implicit conversion of the REAL(4) constants to a REAL(8) data type.

See Also

- Type Declaration Statements
- Type Declarations

Declaration Statements for Character Types

A CHARACTER type specifier can be immediately followed by the length of the character object or function. It takes one of the following forms:

Keyword Forms

CHARACTER [([LEN=]*len*)]

CHARACTER [([LEN=]*len* [, [KIND=]*n*])]

CHARACTER [(KIND=*n* [, LEN=*len*])]

Nonkeyword Form

CHARACTER**len*[,]

len Is one of the following:

- In keyword forms

  The *len* is a specification expression or an asterisk (*). If no length is specified, the default length is 1.

  If the length evaluates to a negative value, the length of the character entity is zero.

- In nonkeyword form

The *len* is a specification expression or an asterisk enclosed in parentheses, or a scalar integer literal constant (with no kind parameter). The comma is permitted only if no double colon (::) appears in the type declaration statement.

This form can also (optionally) be specified following the name of the data object or function (*v***len*). In this case, the length specified overrides any length following the CHARACTER type specifier.

The largest valid value for *len* in both forms is 2**31-1 on IA-32 architecture; 2**63-1 on Intel® 64 architecture and IA-64 architecture. Negative values are treated as zero.

n Is a scalar integer initialization expression specifying a valid kind parameter. Currently the only kind available is 1.

Description

An automatic object can appear in a character declaration. The object cannot be a dummy argument, and its length must be declared with a specification expression that is not a constant expression.

The length specified for a character-valued statement function or statement function dummy argument of type character must be an integer constant expression.

When an asterisk length specification *(*) is used for a function name or dummy argument, it assumes the length of the corresponding function reference or actual argument. Similarly, when an asterisk length specification is used for a named constant, the name assumes the length of the actual constant it represents. For example, STRING assumes a 9-byte length in the following statements:

```
CHARACTER*(*) STRING

PARAMETER (STRING = 'VALUE IS:')
```

A function name must not be declared with a * length, if the function is an internal or module function, or if it is array-valued, pointer-valued, recursive, or pure.

The form CHARACTER*(*) is an obsolescent feature in Fortran 95.

Examples

In the following example, the character string last_name is given a length of 20:

```
CHARACTER (LEN=20) last_name
```

In the following example, stri is given a length of 12, while the other two variables retain a length of 8.

```
CHARACTER *8 strg, strh, stri*12
```

In the following example, as a dummy argument strh is given the length of an assigned string when it is assigned, while the other two variables retain a length of 8:

```
CHARACTER *8 strg, strh(*), stri
```

The following examples show ways to specify strings of known length:

```
CHARACTER*32 string
```

```
CHARACTER string*32
```

The following examples show ways to specify strings of unknown length:

```
CHARACTER string*(*)
```

```
CHARACTER*(*) string
```

The following example declares an array NAMES containing 100 32-character elements, an array SOCSEC containing 100 9-character elements, and a variable NAMETY that is 10 characters long and has an initial value of 'ABCDEFGHIJ'.

```
CHARACTER*32 NAMES(100),SOCSEC(100)*9,NAMETY*10 /'ABCDEFGHIJ'/
```

The following example includes a CHARACTER statement declaring two 8-character variables, LAST and FIRST.

```
INTEGER, PARAMETER :: LENGTH=4
```

```
CHARACTER*(4+LENGTH) LAST, FIRST
```

The following example shows a CHARACTER statement declaring an array LETTER containing 26 one-character elements. It also declares a dummy argument BUBBLE that has a passed length defined by the calling program.

```
CHARACTER LETTER(26), BUBBLE*(*)
```

In the following example, NAME2 is an automatic object:

```
SUBROUTINE AUTO_NAME(NAME1)

  CHARACTER(LEN = *) NAME1

  CHARACTER(LEN = LEN(NAME1)) NAME2
```

See Also

- Type Declaration Statements
- Obsolescent features in Fortran 95

- Data Types of Scalar Variables
- Assumed-Length Character Arguments
- Type Declarations

Declaration Statements for Derived Types

The derived-type (TYPE) declaration statement specifies the properties of objects and functions of derived (user-defined) type.

The derived type must be defined before you can specify objects of that type in a TYPE type declaration statement.

An object of derived type must not have the PUBLIC attribute if its type is PRIVATE.

A structure constructor specifies values for derived-type objects.

Examples

The following are examples of derived-type declaration statements:

```
TYPE(EMPLOYEE) CONTRACT

...

TYPE(SETS), DIMENSION(:,:), ALLOCATABLE :: SUBSET_1
```

The following example shows a public type with private components:

```
TYPE LIST_ITEMS

  PRIVATE

  ...

  TYPE(LIST_ITEMS), POINTER :: NEXT, PREVIOUS

END TYPE LIST_ITEMS
```

See Also

- Type Declaration Statements
- TYPE
- Use and host association
- PUBLIC
- PRIVATE
- Structure constructors
- Type Declarations

Declaration Statements for Arrays

An array declaration (or array declarator) declares the shape of an array. It takes the following form:

(*a-spec*)

a-spec Is one of the following array specifications:

- Explicit-shape
- Assumed-shape
- Assumed-size
- Deferred-shape

The array specification can be appended to the name of the array when the array is declared.

Examples

The following examples show array declarations:

```
SUBROUTINE SUB(N, C, D, Z)

  REAL, DIMENSION(N, 15) :: IARRY       ! An explicit-shape array

  REAL C(:), D(0:)                      ! An assumed-shape array

  REAL, POINTER :: B(:,:)               ! A deferred-shape array pointer

  REAL, ALLOCATABLE, DIMENSION(:) :: K  ! A deferred-shape allocatable array

  REAL :: Z(N,*)                        ! An assumed-size array
```

See Also

- Type Declaration Statements
- Explicit-Shape Specifications
- Assumed-Shape Specifications
- Assumed-Size Specifications
- Deferred-Shape Specifications
- Type Declarations

Explicit-Shape Specifications

An *explicit-shape array* is declared with explicit values for the bounds in each dimension of the array. An explicit-shape specification takes the following form:

([*dl*:] *du*[, [*dl*:] *du*] ...)

dl Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.
If the lower bound is not specified, it is assumed to be 1

du Is a specification expression indicating the upper bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type.
The bounds can be specified as constant or nonconstant expressions, as follows:

- If the bounds are constant expressions, the subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds. If the lower bound is greater than the upper bound, the range is empty, the extent in that dimension is zero, and the array has a size of zero.
- If the bounds are nonconstant expressions, the array must be declared in a procedure. The bounds can have different values each time the procedure is executed, since they are determined when the procedure is entered.

  The bounds are not affected by any redefinition or undefinition of the variables in the specification expression that occurs while the procedure is executing.

  The following explicit-shape arrays can specify nonconstant bounds:

  - An automatic array (the array is a local variable)
  - An adjustable array (the array is a dummy argument to a subprogram)

The following are examples of explicit-shape specifications:

```
INTEGER I(3:8, -2:5)          ! Rank-two array; range of dimension
one is

...                           ! 3 to 8, range of dimension two is -2
 to 5

SUBROUTINE SUB(A, B, C)

  INTEGER :: B, C

  REAL, DIMENSION(B:C) :: A  ! Rank-one array; range is B to C
```

Consider the following:

```
INTEGER M(10, 10, 10)

INTEGER K(-3:6, 4:13, 0:9)
```

M and K are both explicit-shape arrays with a rank of 3, a size of 1000, and the same shape (10,10,10). Array M uses the default lower bound of 1 for each of its dimensions. So, when it is declared only the upper bound needs to be specified. Each of the dimensions of array K has a lower bound other than the default, and the lower bounds as well as the upper bounds are declared.

Automatic Arrays

An *automatic array* is an explicit-shape array that is a local variable. Automatic arrays are only allowed in function and subroutine subprograms, and are declared in the specification part of the subprogram. At least one bound of an automatic array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The following example shows automatic arrays:

```
SUBROUTINE SUB1 (A, B)

  INTEGER A, B, LOWER

  COMMON /BOUND/ LOWER

  ...

  INTEGER AUTO_ARRAY1(B)

  ...

  INTEGER AUTO_ARRAY2(LOWER:B)

  ...

  INTEGER AUTO_ARRAY3(20, B*A/2)

END SUBROUTINE
```

Consider the following:

```
 SUBROUTINE EXAMPLE (N, R1, R2)

   DIMENSION A (N, 5), B(10*N)

   ...

   N = IFIX(R1) + IFIX(R2)
```

When the subroutine is called, the arrays A and B are dimensioned on entry into the subroutine with the value of the passed variable N. Later changes to the value of N have no effect on the dimensions of array A or B.

Adjustable Arrays

An *adjustable array* is an explicit-shape array that is a dummy argument to a subprogram. At least one bound of an adjustable array must be a nonconstant specification expression. The bounds are determined when the subprogram is called.

The array specification can contain integer variables that are either dummy arguments or variables in a common block.

When the subprogram is entered, each dummy argument specified in the bounds must be associated with an actual argument. If the specification includes a variable in a common block, the variable must have a defined value. The array specification is evaluated using the values of the actual arguments, as well as any constants or common block variables that appear in the specification.

The size of the adjustable array must be less than or equal to the size of the array that is its corresponding actual argument.

To avoid possible errors in subscript evaluation, make sure that the bounds expressions used to declare multidimensional adjustable arrays match the bounds as declared by the caller.

In the following example, the function computes the sum of the elements of a rank-two array. Notice how the dummy arguments M and N control the iteration:

```
FUNCTION THE_SUM(A, M, N)
  DIMENSION A(M, N)
  SUMX = 0.0
  DO J = 1, N
    DO I = 1, M
      SUMX = SUMX + A(I, J)
    END DO
  END DO
  THE_SUM = SUMX
END FUNCTION
```

The following are examples of calls on THE_SUM:

```
DIMENSION A1(10,35), A2(3,56)

SUM1 = THE_SUM(A1,10,35)

SUM2 = THE_SUM(A2,3,56)
```

The following example shows how the array bounds determined when the procedure is entered do not change during execution:

```
DIMENSION ARRAY(9,5)

L = 9

M = 5

CALL SUB(ARRAY,L,M)

END

SUBROUTINE SUB(X,I,J)

  DIMENSION X(-I/2:I/2,J)

  X(I/2,J) = 999

  J = 1

  I = 2

END
```

The assignments to I and J do not affect the declaration of adjustable array X as X(-4:4,5) on entry to subroutine SUB.

See Also

- Declaration Statements for Arrays
- Specification expressions

Assumed-Shape Specifications

An *assumed-shape array* is a dummy argument array that assumes the shape of its associated actual argument array. An assumed-shape specification takes the following form:

([*dl*]:[, [*dl*]:] ...)

dl	Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.

The rank of the array is the number of colons (:) specified.

The value of the upper bound is the extent of the corresponding dimension of the associated actual argument array + *lower-bound* - 1.

Examples

The following is an example of an assumed-shape specification:

```
INTERFACE
  SUBROUTINE SUB(M)
    INTEGER M(:, 1:, 5:)
  END SUBROUTINE
END INTERFACE
INTEGER L(20, 5:25, 10)
CALL SUB(L)
SUBROUTINE SUB(M)
  INTEGER M(:, 1:, 5:)
END SUBROUTINE
```

Array M has the same extents as array L, but array M has bounds (1:20, 1:21, 5:14).

Note that an explicit interface is *required* when calling a routine that expects an assumed-shape or pointer array.

Consider the following:

```
 SUBROUTINE ASSUMED(A)
   REAL A(:, :, :)
```

Array A has rank 3, indicated by the three colons (:) separated by commas (,). However, the extent of each dimension is unspecified. When the subroutine is called, A takes its shape from the array passed to it. For example, consider the following:

```
 REAL X (4, 7, 9)
 ...
 CALL ASSUMED(X)
```

This gives A the dimensions (4, 7, 9). The actual array and the assumed-shape array must have the same rank.

Consider the following:

```
SUBROUTINE ASSUMED(A)
  REAL A(3:, 0:, -2:)
...
```

If the subroutine is called with the same actual array X(4, 7, 9), as in the previous example, the lower and upper bounds of A would be:

```
A(3:6, 0:6, -2:6)
```

Assumed-Size Specifications

An *assumed-size array* is a dummy argument array that assumes the size (only) of its associated actual argument array; the rank and extents can differ for the actual and dummy arrays. An assumed-size specification takes the following form:

([*expli-shape-spec*,] [*expli-shape-spec*,] ... [*dl*:] *)

expli-shape-spec	Is an explicit-shape specification.
dl	Is a specification expression indicating the lower bound of the dimension. The expression can have a positive, negative, or zero value. If necessary, the value is converted to integer type. If the lower bound is not specified, it is assumed to be 1.
*	Is the upper bound of the last dimension.

The rank of the array is the number of explicit-shape specifications plus 1.

The size of the array is assumed from the actual argument associated with the assumed-size dummy array as follows:

- If the actual argument is an array of type other than default character, the size of the dummy array is the size of the actual array.
- If the actual argument is an array element of type other than default character, the size of the dummy array is `a + 1 - s`, where *s* is the subscript order value and *a* is the size of the actual array.
- If the actual argument is a default character array, array element, or array element substring, and it begins at character storage unit *b* of an array with *n* character storage units, the size of the dummy array is as follows:

  ```
  MAX(INT((n + 1 - b)/y), 0)
  ```

  The *y* is the length of an element of the dummy array.

An assumed-size array can only be used as a whole array reference in the following cases:

- When it is an actual argument in a procedure reference that does not require the shape

- In the intrinsic function LBOUND

Because the actual size of an assumed-size array is unknown, an assumed-size array cannot be used as any of the following in an I/O statement:

- An array name in the I/O list
- A unit identifier for an internal file
- A run-time format specifier

Examples

The following is an example of an assumed-size specification:

```
SUBROUTINE SUB(A, N)

  REAL A, N

  DIMENSION A(1:N, *)

  ...
```

The following example shows that you can specify lower bounds for any of the dimensions of an assumed-size array, including the last:

```
SUBROUTINE ASSUME(A)

  REAL A(-4:-2, 4:6, 3:*)
```

See Also

- Declaration Statements for Arrays
- Array Elements

Deferred-Shape Specifications

A *deferred-shape array* is an array pointer or an allocatable array.

The array specification contains a colon (:) for each dimension of the array. No bounds are specified. The bounds (and shape) of allocatable arrays and array pointers are determined when space is allocated for the array during program execution.

An *array pointer* is an array declared with the POINTER attribute. Its bounds and shape are determined when it is associated with a target by pointer assignment, or when the pointer is allocated by execution of an ALLOCATE statement.

In pointer assignment, the lower bound of each dimension of the array pointer is the result of the LBOUND intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND intrinsic function applied to the corresponding dimension of the target.

A pointer dummy argument can be associated only with a pointer actual argument. An actual argument that is a pointer can be associated with a nonpointer dummy argument.

A function result can be declared to have the pointer attribute.

An *allocatable array* is declared with the ALLOCATABLE attribute. Its bounds and shape are determined when the array is allocated by execution of an ALLOCATE statement.

Examples

The following are examples of deferred-shape specifications:

```
REAL, ALLOCATABLE :: A(:,:)       ! Allocatable array
REAL, POINTER :: C(:), D (:,:,:)  ! Array pointers
```

If a deferred-shape array is declared in a DIMENSION or TARGET statement, it must be given the ALLOCATABLE or POINTER attribute in another statement. For example:

```
DIMENSION P(:, :, :)
POINTER P
TARGET B(:,:)
ALLOCATABLE B
```

If the deferred-shape array is an array of pointers, its size, shape, and bounds are set in an ALLOCATE statement or in the pointer assignment statement when the pointer is associated with an allocated target. A pointer and its target must have the same rank.

For example:

```
REAL, POINTER :: A(:,:), B(:), C(:,:)
INTEGER, ALLOCATABLE :: I(:)
REAL, ALLOCATABLE, TARGET :: D(:, :), E(:)
      ...
ALLOCATE (A(2, 3), I(5), D(SIZE(I), 12), E(98) )
C => D              ! Pointer assignment statement
B => E(25:56)       ! Pointer assignment to a section
           !   of a target
```

See Also

- Declaration Statements for Arrays
- POINTER

- ALLOCATABLE
- ALLOCATE
- Pointer assignment
- LBOUND
- UBOUND

ALLOCATABLE Attribute and Statement Overview

The ALLOCATABLE attribute specifies that an array is an allocatable array with a deferred shape. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array. For more information, see ALLOCATABLE.

ASYNCHRONOUS Attribute and Statement Overview

The ASYNCHRONOUS attribute specifies that a variable can be used for asynchronous input and output. For more information, see ASYNCHRONOUS.

AUTOMATIC and STATIC Attributes and Statements Overview

The AUTOMATIC and STATIC attributes control the storage allocation of variables in subprograms. For more information, see AUTOMATIC and STATIC.

BIND Attribute and Statement Overview

The BIND statement specifies that an object is interoperable with C and has external linkage. For more information, see BIND.

COMMON Statement Overview

A COMMON statement defines one or more contiguous areas, or blocks, of physical storage (called common blocks) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items. For more information, see COMMON.

DATA Statement Overview

The DATA statement assigns initial values to variables before program execution. For more information, see DATA.

DIMENSION Attribute and Statement Overview

The DIMENSION attribute specifies that an object is an array, and defines the shape of the array. For more information, see DIMENSION.

EQUIVALENCE Statement Overview

The EQUIVALENCE statement specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area. For more information, see EQUIVALENCE.

See Also

- Specification Statements
- Making Arrays Equivalent
- Making Substrings Equivalent
- EQUIVALENCE and COMMON Interaction
- Making Arrays Equivalent
- Making Substrings Equivalent
- EQUIVALENCE and COMMON Interaction

Making Arrays Equivalent

When you make an element of one array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between the other elements of the two arrays. Thus, if the first elements of two equal-sized arrays are made equivalent, both arrays share the same storage. If the third element of a 7-element array is made equivalent to the first element of another array, the last five elements of the first array overlap the first five elements of the second array.

Two or more elements of the same array should not be associated with each other in one or more EQUIVALENCE statements. For example, you cannot use an EQUIVALENCE statement to associate the first element of one array with the first element of another array, and then attempt to associate the fourth element of the first array with the seventh element of the other array.

Consider the following example:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE(TABLE(2,2), TRIPLE(1,2,2))
```

These statements cause the entire array TABLE to share part of the storage allocated to TRIPLE. The following table shows how these statements align the arrays:

Table 576: Equivalence of Array Storage

Array TRIPLE		Array TABLE	
Array Element	**Element Number**	**Array Element**	**Element Number**
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7	TABLE(2,2)	4
TRIPLE(2,2,2)	8		

Each of the following statements also aligns the two arrays as shown in the above table:

```
EQUIVALENCE(TABLE, TRIPLE(2,2,1))
EQUIVALENCE(TRIPLE(1,1,2), TABLE(2,1))
```

You can also make arrays equivalent with nonunity lower bounds. For example, an array defined as A(2:3,4) is a sequence of eight values. A reference to A(2,2) refers to the third element in the sequence. To make array A(2:3,4) share storage with array B(2:4,4), you can use the following statement:

```
EQUIVALENCE(A(3,4), B(2,4))
```

The entire array A shares part of the storage allocated to array B. The following table shows how these statements align the arrays. The arrays can also be aligned by the following statements:

```
EQUIVALENCE(A, B(4,1))
EQUIVALENCE(B(3,2), A(2,2))
```

Table 577: Equivalence of Arrays with Nonunity Lower Bounds

Array B		Array A	
Array Element	**Element Number**	**Array Element**	**Element Number**
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Only in the EQUIVALENCE statement can you identify an array element with a single subscript (the linear element number), even though the array was defined as multidimensional. For example, the following statements align the two arrays as shown in the above table:

```
DIMENSION B(2:4,1:4), A(2:3,1:4)
EQUIVALENCE(B(6), A(4))
```

Making Substrings Equivalent

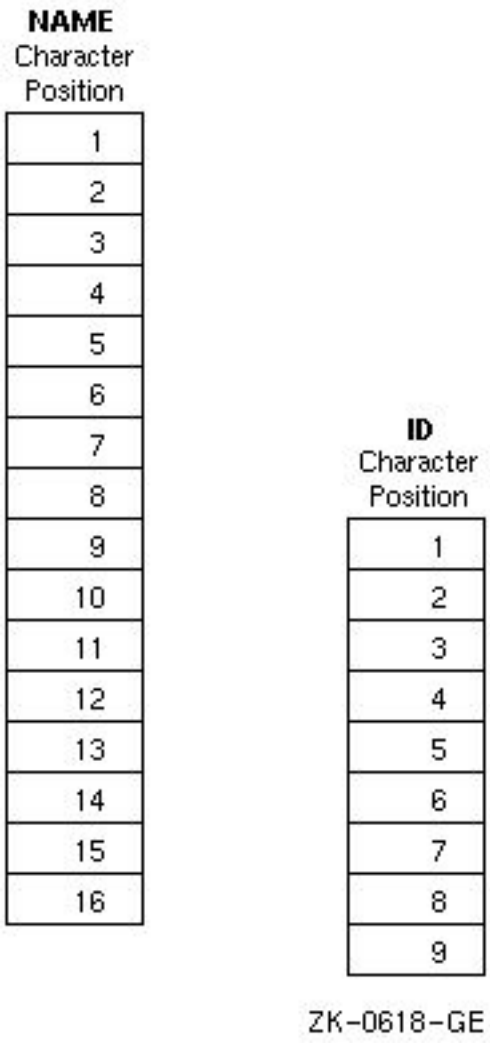
When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets associations between the other corresponding characters in the character entities; for example:

```
CHARACTER NAME*16, ID*9
EQUIVALENCE(NAME(10:13), ID(2:5))
```

These statements cause character variables NAME and ID to share space (see the following figure). The arrays can also be aligned by the following statement:

```
EQUIVALENCE(NAME(9:9), ID(1:1))
```

Figure 28: Equivalence of Substrings



If the character substring references are array elements, the EQUIVALENCE statement sets associations between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position. For example, the following statements cause character arrays FIELDS and STAR to share storage (see the following figure).

```
CHARACTER FIELDS(100)*4, STAR(5)*5

EQUIVALENCE(FIELDS(1)(2:4), STAR(2)(3:5))
```

Figure 29: Equivalence of Character Arrays

The EQUIVALENCE statement cannot assign the same storage location to two or more substrings that start at different character positions in the same character variable or character array. The EQUIVALENCE statement also cannot assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays.

EQUIVALENCE and COMMON Interaction

A common block can extend beyond its original boundaries if variables or arrays are associated with entities stored in the common block. However, a common block can only extend beyond its last element; the extended portion cannot precede the first element in the block.

Examples

The following two figures demonstrate valid and invalid extensions of the common block, respectively.

Figure 30: A Valid Extension of a Common Block

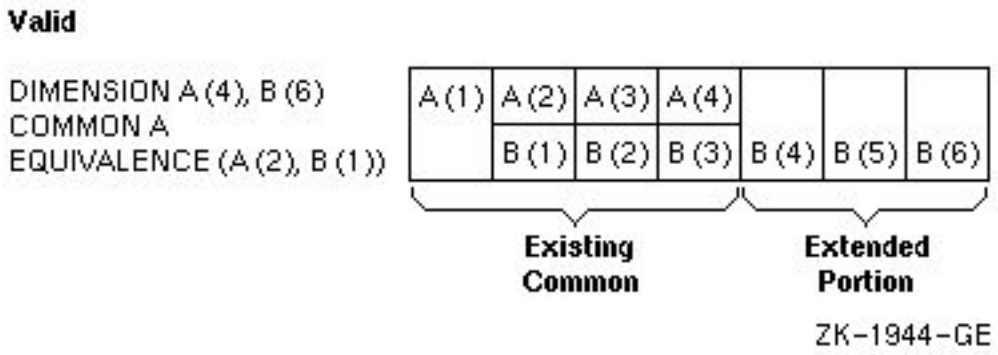
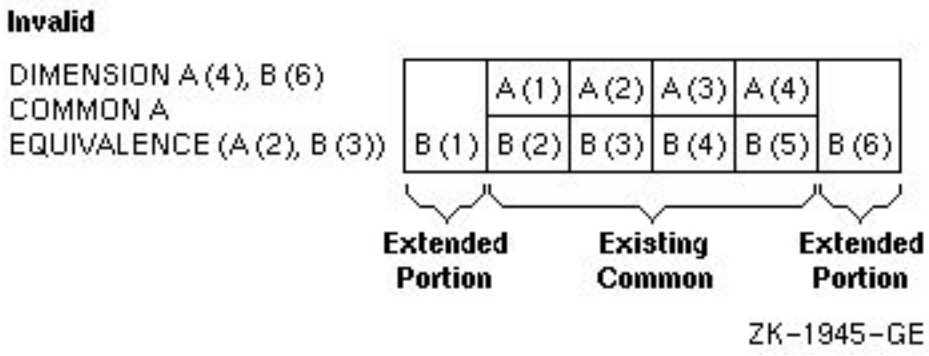


Figure 31: An Invalid Extension of a Common Block



The second example is invalid because the extended portion, B(1), precedes the first element of the common block.

The following example shows a valid EQUIVALENCE statement and an invalid EQUIVALENCE statement in the context of a common block.

```
COMMON A, B, C

DIMENSION D(3)

EQUIVALENCE(B, D(1))      ! Valid, because common block is extended

                ! from the end.

COMMON A, B, C

DIMENSION D(3)

EQUIVALENCE(B, D(3))      ! Invalid, because D(1) would extend common

                ! block to precede A's location.
```

EXTERNAL Attribute and Statement Overview

The EXTERNAL attribute allows an external or dummy procedure to be used as an actual argument. For more information, see EXTERNAL.

IMPLICIT Statement Overview

The IMPLICIT statement overrides the default implicit typing rules for names. For more information, see IMPLICIT.

INTENT Attribute and Statement Overview

The INTENT attribute specifies the intended use of one or more dummy arguments. For more information, see INTENT.

INTRINSIC Attribute and Statement Overview

The INTRINSIC attribute allows the specific name of an intrinsic procedure to be used as an actual argument. Certain specific function names cannot be used; these are indicated in table Intrinsic Functions Not Allowed as Actual Arguments.

For more information, see INTRINSIC.

NAMELIST Statement Overview

The NAMELIST statement associates a name with a list of variables. This group name can be referenced in some input/output operations. For more information, see NAMELIST.

OPTIONAL Attribute and Statement Overview

The OPTIONAL attribute permits dummy arguments to be omitted in a procedure reference. For more information, see OPTIONAL.

PARAMETER Attribute and Statement Overview

The PARAMETER attribute defines a named constant. For more information, see PARAMETER.

POINTER Attribute and Statement Overview

The POINTER attribute specifies that an object is a pointer (a dynamic variable). For more information, see POINTER.

PROTECTED Attribute and Statement Overview

The PROTECTED attribute specifies limitations on the use of module entities. For more information, see PROTECTED.

PUBLIC and PRIVATE Attributes and Statements Overview

The PRIVATE and PUBLIC attributes specify the accessibility of entities in a module. (These attributes are also called accessibility attributes.) For more information, see PUBLIC and PRIVATE.

SAVE Attribute and Statement Overview

The SAVE attribute causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram. For more information, see SAVE.

TARGET Attribute and Statement Overview

The TARGET attribute specifies that an object can become the target of a pointer. For more information, see TARGET.

IMPORT Statement Overview

The IMPORT statement makes host entities accessible in the interface body of an interface block. For more information, see IMPORT.

VOLATILE Attribute and Statement Overview

The VOLATILE attribute specifies that the value of an object is entirely unpredictable, based on information local to the current program unit. For more information, see VOLATILE.

47 Dynamic Allocation

Data objects can be static or dynamic. If a data object is static, a fixed amount of memory storage is created for it at compile time and is not freed until the program exits. If a data object is dynamic, memory storage for the object can be created (allocated), altered, or freed (deallocated) as a program executes.

In Fortran 95/90, pointers, allocatable arrays, and automatic arrays are dynamic data objects.

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target. A pointer can be dynamically disassociated from a target by using a NULLIFY statement.

An ALLOCATE statement can also be used to create storage for an allocatable array. A DEALLOCATE statement is used to free the storage space reserved in a previous ALLOCATE statement.

Automatic arrays differ from allocatable arrays in that they are automatically allocated and deallocated whenever you enter or leave a procedure, respectively.

NOTE. Dynamic memory allocation is limited by several factors, including swap file size and memory requirements of other applications that are running. Dynamic allocations that are too large or otherwise attempt to use the protected memory of other applications result in General Protection Fault errors. If you encounter an unexpectedly low limit, you might need to reset your virtual memory size through the Control Panel or redefine the swap file size.

Some programming techniques can help minimize memory requirements, such as using one large array instead of two or more individual arrays. Allocated arrays that are no longer needed should be deallocated.

ALLOCATE Statement Overview

The ALLOCATE statement dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized. For more information, see ALLOCATE.

See Also

- Dynamic Allocation
- Allocation of Allocatable Arrays
- Allocation of Pointer Targets
- Allocation of Allocatable Arrays
- Allocation of Pointer Targets

Allocation of Allocatable Arrays

The bounds (and shape) of an allocatable array are determined when it is allocated. Subsequent redefinition or undefinition of any entities in the bound expressions does not affect the array specification.

If the lower bound is greater than the upper bound, that dimension has an extent of zero, and the array has a size of zero. If the lower bound is omitted, it is assumed to be 1.

When an array is allocated, it is definable. If you try to allocate a currently allocated allocatable array, an error occurs.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
REAL, ALLOCATABLE :: E(:,:)

...

IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7))
```

Allocation Status

During program execution, the allocation status of an allocatable array is one of the following:

- Not currently allocated

  The array was never allocated or the last operation on it was a deallocation. Such an array must not be referenced or defined.

- Currently allocated

  The array was allocated by an ALLOCATE statement. Such an array can be referenced, defined, or deallocated.

If an allocatable array has the SAVE attribute, it has an initial status of "not currently allocated". If the array is then allocated, its status changes to "currently allocated". It keeps that status until the array is deallocated.

If an allocatable array *does not* have the SAVE attribute, it has the status of "not currently allocated" at the beginning of each invocation of the procedure. If the array's status changes to "currently allocated", it is deallocated if the procedure is terminated by execution of a RETURN or END statement.

Example: Allocating Virtual Memory

The following example shows a program that performs virtual memory allocation. This program uses Fortran 95/90 standard-conforming statements instead of calling an operating system memory allocation routine.

```
! Program accepts an integer and displays square root values
  INTEGER(4) :: N
  READ (5,*) N                          ! Reads an integer value
  CALL MAT(N)
  END
! Subroutine MAT uses the typed integer value to display the square
! root values of numbers from 1 to N (the number read)
  SUBROUTINE MAT(N)
  REAL(4), ALLOCATABLE :: SQR(:)        ! Declares SQR as a one-dimensional
                                        !          allocatable array
  ALLOCATE (SQR(N))                     ! Allocates array SQR
  DO J=1,N
     SQR(J) = SQRT(FLOATJ(J))           ! FLOATJ converts integer to REAL
  ENDDO
  WRITE (6,*) SQR                       ! Displays calculated values
  DEALLOCATE (SQR)                      ! Deallocates array SQR
  END SUBROUTINE MAT
```

See Also

- ALLOCATE Statement Overview
- ALLOCATED intrinsic function
- ALLOCATE statement

Allocation of Pointer Targets

When a pointer is allocated, the pointer is associated with a target and can be used to reference or define the target. (The target can be an array or a scalar, depending on how the pointer was declared.)

Other pointers can become associated with the pointer target (or part of the pointer target) by pointer assignment.

In contrast to allocatable arrays, a pointer can be allocated a new target even if it is currently associated with a target. The previous association is broken and the pointer is then associated with the new target.

If the previous target was created by allocation, it becomes inaccessible unless it can still be referred to by other pointers that are currently associated with it.

The intrinsic function ASSOCIATED can be used to determine whether a pointer is currently associated with a target. (The association status of the pointer must be *defined*.) For example:

```
REAL, TARGET  :: TAR(0:50)

REAL, POINTER :: PTR(:)

PTR => TAR

...

IF (ASSOCIATED(PTR,TAR))...
```

See Also

- ALLOCATE Statement Overview
- POINTER statement and attribute
- Pointer assignments
- ASSOCIATED intrinsic function

DEALLOCATE Statement Overview

The DEALLOCATE statement frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated). For more information, see DEALLOCATE.

See Also

- Dynamic Allocation
- Deallocation of Allocatable Arrays
- Deallocation of Pointer Targets
- Deallocation of Allocatable Arrays
- Deallocation of Pointer Targets

Deallocation of Allocatable Arrays

If the DEALLOCATE statement specifies an array that is not currently allocated, an error occurs.

If an allocatable array with the TARGET attribute is deallocated, the association status of any pointer associated with it becomes undefined.

If a RETURN or END statement terminates a procedure, an allocatable array has one of the following allocation statuses:

- It keeps its previous allocation and association status if the following is true:
  - It has the SAVE attribute.
  - It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
  - It is accessible by host association.
- It remains allocated if it is accessed by use association.
- Otherwise, its allocation status is deallocated.

The intrinsic function ALLOCATED can be used to determine whether an allocatable array is currently allocated; for example:

```
SUBROUTINE TEST

  REAL, ALLOCATABLE, SAVE :: F(:,:)

  REAL, ALLOCATABLE :: E(:,:,:)

  ...

  IF (.NOT. ALLOCATED(E)) ALLOCATE(E(2:4,7,14))

END SUBROUTINE TEST
```

Note that when subroutine TEST is exited, the allocation status of F is maintained because F has the SAVE attribute. Since E does not have the SAVE attribute, it is deallocated. On the next invocation of TEST, E will have the status of "not currently allocated".

See Also

- DEALLOCATE Statement Overview
- Host association
- TARGET statement and attribute
- RETURN statement
- END statement
- SAVE statement

Deallocation of Pointer Targets

A pointer must not be deallocated unless it has a defined association status. If the DEALLOCATE statement specifies a pointer that has undefined association status, or a pointer whose target was not created by allocation, an error occurs.

A pointer must not be deallocated if it is associated with an allocatable array, or it is associated with a portion of an object (such as an array element or an array section).

If a pointer is deallocated, the association status of any other pointer associated with the target (or portion of the target) becomes undefined.

Execution of a RETURN or END statement in a subprogram causes the pointer association status of any pointer declared (or accessed) in the procedure to become undefined, unless any of the following applies to the pointer:

- It has the SAVE attribute.
- It is in the scoping unit of a module that is accessed by another scoping unit which is currently executing.
- It is accessible by host association.
- It is in blank common.
- It is in a named common block that appears in another scoping unit that is currently executing.
- It is the return value of a function declared with the POINTER attribute.

If the association status of a pointer becomes undefined, it cannot subsequently be referenced or defined.

Examples

The following example shows deallocation of a pointer:

```
INTEGER ERR

REAL, POINTER :: PTR_A(:)

...

ALLOCATE (PTR_A(10), STAT=ERR)

...

DEALLOCATE(PTR_A)
```

See Also

- DEALLOCATE Statement Overview

- POINTER statement and attribute
- COMMON statement
- NULL intrinsic function
- Host association
- TARGET statement and attribute
- RETURN statement
- END statement
- SAVE statement

NULLIFY Statement Overview

The NULLIFY statement disassociates a pointer from its target. For more information, see NULLIFY.

Execution Control

48

A program normally executes statements in the order in which they are written. Executable control constructs and statements modify this normal execution by transferring control to another statement in the program, or by selecting blocks (groups) of constructs and statements for execution or repetition.

In Fortran 95/90, control constructs (CASE, DO, and IF) can be named. The name must be a unique identifier in the scoping unit, and must appear on the initial line and terminal line of the construct. On the initial line, the name is separated from the statement keyword by a colon (:).

A block can contain any executable Fortran statement except an END statement. You can transfer control out of a block, but you cannot transfer control into another block.

DO loops cannot partially overlap blocks. The DO statement and its terminal statement must appear together in a statement block.

Branch Statements

Branching affects the normal execution sequence by transferring control to a labeled statement in the same scoping unit. The transfer statement is called the *branch statement*, while the statement to which the transfer is made is called the *branch target statement*.

Any executable statement can be a branch target statement, except for the following:

- CASE statement
- ELSE statement
- ELSE IF statement

Certain restrictions apply to the following statements:

Statement	Restriction
DO terminal statement	The branch must be taken from within its nonblock DO construct[1].
END DO	The branch must be taken from within its block DO construct.
END IF	The branch should be taken from within its IF construct[2].
END SELECT	The branch must be taken from within its CASE construct.

Statement	Restriction

[1] If the terminal statement is shared by more than one nonblock DO construct, the branch can only be taken from within the innermost DO construct

[2] You can branch to an END IF statement from outside the IF construct; this is a deleted feature in Fortran 95. Intel® Fortran fully supports features deleted in Fortran 95.

See Also

- Execution Control
- Unconditional GO TO Statement Overview
- Computed GO TO Statement Overview
- The ASSIGN and Assigned GO TO Statements Overview
- Arithmetic IF Statement Overview
- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- IF constructs
- CASE constructs
- DO constructs

Unconditional GO TO Statement Overview

The unconditional GO TO statement transfers control to the same branch target statement every time it executes. For more information, see GOTO - Unconditional.

Computed GO TO Statement Overview

The computed GO TO statement transfers control to one of a set of labeled branch target statements based on the value of an expression. For more information, see GOTO - COMPUTED.

The ASSIGN and Assigned GO TO Statements Overview

The ASSIGN statement assigns a label to an integer variable. Subsequently, this variable can be used as a branch target statement by an assigned GO TO statement or as a format specifier in a formatted input/output statement.

The ASSIGN and assigned GO TO statements are deleted features in Fortran 95; they were obsolescent features in Fortran 90. Intel® Fortran fully supports features deleted in Fortran 95.

Arithmetic IF Statement Overview

The arithmetic IF statement conditionally transfers control to one of three statements, based on the value of an arithmetic expression. For more information, see IF - Arithmetic.

CALL Statement Overview

The CALL statement transfers control to a subroutine subprogram. For more information, see CALL.

CASE Constructs Overview

The CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement. For more information, see CASE.

CONTINUE Statement Overview

The CONTINUE statement is primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement. For more information, see CONTINUE.

DO Constructs Overview

The DO construct controls the repeated execution of a block of statements or constructs. For more information, see DO.

See Also

- Execution Control
- Forms for DO Constructs
- Execution of DO Constructs
- DO WHILE Statement Overview
- CYCLE Statement Overview
- EXIT Statement Overview
- Forms for DO Constructs
- Execution of DO Constructs
- DO WHILE Statement
- CYCLE Statement

- EXIT Statement

Forms for DO Constructs

A DO construct can be in block or nonblock form. For more information, see DO.

Execution of DO Constructs

The range of a DO construct includes all the statements and constructs that follow the DO statement, up to and including the terminal statement. If the DO construct contains another construct, the inner (nested) construct must be entirely contained within the DO construct.

Execution of a DO construct differs depending on how the loop is controlled, as follows:

- For simple DO constructs, there is no loop control. Statements in the DO range are repeated until the DO statement is terminated explicitly by a statement within the range.
- For iterative DO statements, loop control is specified as *do-var = expr1, expr2 [,expr3]*. An iteration count specifies the number of times the DO range is executed. (For more information, see Iteration Loop Control.)
- For DO WHILE statements, loop control is specified as a DO range. The DO range is repeated as long as a specified condition remains true. Once the condition is evaluated as false, the DO construct terminates. (For more information, see the DO WHILE statement.)

See Also

- DO Constructs Overview
- Iteration Loop Control
- Nested DO Constructs
- Extended Range
- Nested DO Constructs
- Extended Range

Iteration Loop Control

DO iteration loop control takes the following form:

do-var = *expr1*, *expr2* [, *expr3*]

do-var	Is the name of a scalar variable of type integer or real. It cannot be the name of an array element or structure component.
expr	Is a scalar numeric expression of type integer, logical, or real. If it is not the same type as *do-var*, it is converted to that type.

Description

A DO variable or expression of type real is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel® Fortran fully supports features deleted in Fortran 95.

The following steps are performed in iteration loop control:

1. The expressions *expr1*, *expr2*, and *expr3* are evaluated to respectively determine the initial, terminal, and increment parameters.

   The increment parameter (*expr3*) is optional and must not be zero. If an increment parameter is not specified, it is assumed to be of type default integer with a value of 1.
2. The DO variable (*do-var*) becomes defined with the value of the initial parameter (*expr1*).
3. The iteration count is determined as follows:

   ```
   MAX(INT((expr2 - expr1 + expr3)/expr3), 0)
   ```

   The iteration count is zero if either of the following is true:

   ```
   expr1 > expr2 and expr3 > 0
   ```

   ```
   expr1 < expr2 and expr3 < 0
   ```
4. The iteration count is tested. If the iteration count is zero, the loop terminates and the DO construct becomes inactive. (Compiler option f66 can affect this.) If the iteration count is nonzero, the range of the loop is executed.
5. The iteration count is decremented by one, and the DO variable is incremented by the value of the increment parameter, if any.

After termination, the DO variable retains its last value (the one it had when the iteration count was tested and found to be zero).

The DO variable must not be redefined or become undefined during execution of the DO range.

If you change variables in the initial, terminal, or increment expressions during execution of the DO construct, it does not affect the iteration count. The iteration count is fixed each time the DO construct is entered.

Examples

The following example specifies 25 iterations:

```
DO 100 K=1,50,2
```

K=49 during the final iteration, K=51 after the loop.

The following example specifies 27 iterations:

```
DO 350 J=50,-2,-2
```

J=-2 during the final iteration, J=-4 after the loop.

The following example specifies 9 iterations:

```
DO NUMBER=5,40,4
```

NUMBER=37 during the final iteration, NUMBER=41 after the loop. The terminating statement of this DO loop must be END DO.

See Also

- Execution of DO Constructs
- Obsolescent and Deleted Language Features
- f66

Nested DO Constructs

A DO construct can contain one or more complete DO constructs (loops). The range of an inner nested DO construct must lie completely within the range of the next outer DO construct. Nested nonblock DO constructs can share a labeled terminal statement.

The following figure shows correctly and incorrectly nested DO constructs:

Figure 32: Nested DO Constructs

In a nested DO construct, you can transfer control from an inner construct to an outer construct. However, you cannot transfer control from an outer construct to an inner construct.

If two or more nested DO constructs share the same terminal statement, you can transfer control to that statement only from within the range of the innermost construct. Any other transfer to that statement constitutes a transfer from an outer construct to an inner construct, because the shared statement is part of the range of the innermost construct.

Extended Range

A DO construct has an extended range if both of the following are true:

- The DO construct contains a control statement that transfers control out of the construct.
- Another control statement returns control back into the construct after execution of one or more statements.

The range of the construct is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the construct.

The following rules apply to a DO construct with extended range:

- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.
- The extended range of a DO statement must not change the control variable of the DO statement.

The following figure shows valid and invalid extended range control transfers:

1892

Figure 33: Control Transfers and Extended Range

DO WHILE Statement Overview

The DO WHILE statement executes the range of a DO construct while a specified condition remains true. For more information, see DO WHILE.

CYCLE Statement Overview

The CYCLE statement interrupts the current execution cycle of the innermost (or named) DO construct. For more information, see CYCLE.

EXIT Statement Overview

The EXIT statement terminates execution of a DO construct. For more information, see EXIT.

END Statement Overview

The END statement marks the end of a program unit. For more information, see END.

IF Construct and Statement Overview

The IF construct conditionally executes one block of statements or constructs.

The IF statement conditionally executes one statement.

The decision to transfer control or to execute the statement or block is based on the evaluation of a logical expression within the IF statement or construct.

IF Construct Overview

The IF construct conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. For more information, see IF construct.

IF Statement Overview

The IF statement conditionally executes one statement based on the value of a logical expression. For more information, see IF - Logical.

PAUSE Statement Overview

The PAUSE statement temporarily suspends program execution until the user or system resumes execution. For more information, see PAUSE.

The PAUSE statement is a deleted feature in Fortran 95; it was an obsolescent feature in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.

See Also

- Execution Control

For alternate methods of pausing while reading from and writing to a device, see READ and WRITE.

RETURN Statement Overview

The RETURN statement transfers control from a subprogram to the calling program unit. For more information, see RETURN.

STOP Statement Overview

The STOP statement terminates program execution before the end of the program unit. For more information, see STOP.

Program Units and Procedures 49

A Fortran 95/90 program consists of one or more program units. There are four types of program units:

- Main program

  The program unit that denotes the beginning of execution. It may or may not have a PROGRAM statement as its first statement.
- External procedures

  Program units that are either user-written functions or subroutines.
- Modules

  Program units that contain declarations, type definitions, procedures, or interfaces that can be shared by other program units.
- Block data program units

  Program units that provide initial values for variables in named common blocks.

A program unit does not have to contain executable statements; for example, it can be a module containing interface blocks for subroutines.

A procedure can be invoked during program execution to perform a specific task. There are several kinds of procedures, as follows:

Kind of Procedure	Description
External Procedure	A procedure that is not part of any other program unit.
Module Procedure	A procedure defined within a module
Internal Procedure[1]	A procedure (other than a statement function) contained within a main program, function, or subroutine
Intrinsic Procedure	A procedure defined by the Fortran language
Dummy Procedure	A dummy argument specified as a procedure or appearing in a procedure reference
Statement function	A computing procedure defined by a single statement

Kind of Procedure	Description

[1] The program unit that contains an internal procedure is called its *host*.

A *function* is invoked in an expression using the name of the function or a defined operator. It returns a a single value (function result) that is used to evaluate the expression.

A *subroutine* is invoked in a CALL statement or by a defined assignment statement. It does not directly return a value, but values can be passed back to the calling program unit through arguments (or variables) known to the calling program.

Recursion (direct or indirect) is permitted for functions and subroutines.

A procedure interface refers to the properties of a procedure that interact with or are of concern to the calling program. A procedure interface can be explicitly defined in interface blocks. All program units, except block data program units, can contain interface blocks.

Main Program

A main program is a program unit whose first statement is not a SUBROUTINE, FUNCTION, MODULE, or BLOCK DATA statement. Program execution always begins with the first executable statement in the main program, so there must be exactly one main program unit in every executable program. For more information, see PROGRAM.

Modules and Module Procedures Overview

A module program unit contains specifications and definitions that can be made accessible to other program units. There are two types of modules, intrinsic and nonintrinsic. Intrinsic modules are included in the Fortran library; nonintrinsic modules are user-defined.

For the module to be accessible, the other program units must reference its name in a USE statement, and the module entities must be public. For more information, see MODULE.

A module procedure is a procedure declared and defined in a module, between its CONTAINS and END statements. For more information, see MODULE PROCEDURE.

See Also

- Program Units and Procedures
- Module References
- USE Statement Overview
- Module References
- USE Statement

Module References

A program unit references a module in a USE statement. This module reference lets the program unit access the public definitions, specifications, and procedures in the module.

Entities in a module are public by default, unless the USE statement specifies otherwise or the PRIVATE attribute is specified for the module entities.

A module reference causes use association between the using program unit and the entities in the module.

See Also

- Modules and Module Procedures Overview
- USE statement
- PRIVATE attribute
- PUBLIC attribute
- Use association

USE Statement Overview

The USE statement gives a program unit accessibility to public entities in a module. For more information, see USE.

Examples

Entities in modules can be accessed either through their given name, or through aliases declared in the USE statement of the main program unit. For example:

```
USE MODULE_LIB, XTABS => CROSSTABS
```

This statement accesses the routine called `CROSSTABS` in `MODULE_LIB` by the name `XTABS`. This way, if two modules have routines called `CROSSTABS`, one program can use them both simultaneously by assigning a local name in its USE statement.

When a program or subprogram renames a module entity, the local name (`XTABS`, in the preceding example) is accessible throughout the scope of the program unit that names it.

The ONLY option also allows public variables to be renamed. Consider the following:

```
USE MODULE_A, ONLY: VARIABLE_A => VAR_A
```

In this case, the host program accesses only `VAR_A` from module `A`, and refers to it by the name `VARIABLE_A`.

Consider the following example:

```
MODULE FOO
   integer foos_integer
 PRIVATE
   integer foos_my_integer
END MODULE FOO
```

PRIVATE, in this case, makes the PRIVATE attribute the default for the entire module `FOO`. To make `foos_integer` accessible to other program units, add the line:

```
PUBLIC :: foos_integer
```

Alternatively, to make only `foos_my_integer` inaccessible outside the module, rewrite the module as follows:

```
MODULE FOO
  integer foos_integer
  integer, private::foos_my_integer
END MODULE FOO
```

Intrinsic Modules

Intrinsic modules, like other module program units, contain specifications and definitions that can be made accessible to other program units. The intrinsic modules are part of the Fortran library. They are Fortran 2003 features.

An intrinsic module is specified in a USE statement, as follows:

`USE, INTRINSIC :: mod-name [, rename-list] ...`

`USE, INTRINSIC :: mod-name, ONLY : [, only-list]`

mod-name	Is the name of the intrinsic module.
rename-list	See the description in USE.
only-list	See the description in USE.

Procedures and types defined in an intrinsic module are not themselves intrinsic.

An intrinsic module can have the same name as other global entities, such as program units, common blocks, or external procedures. A scoping unit must not be able to access both an intrinsic module and a non-intrinsic module with the same name.

The following intrinsic modules are included in the Fortran library:

- ISO_C_BINDING
- ISO_FORTRAN_ENV
- IEEE Intrinsic Modules

ISO_C_BINDING Module

The ISO_C_BINDING intrinsic module provides access to data entities that are useful in mixed-language programming. It takes the following form:

```
USE, INTRINSIC :: ISO_C_BINDING
```

This intrinsic module provides access to the following data entities:

- Named Constants
- Derived Types

  Derived type C_PTR is interoperable with any C object pointer type. Derived type C_FUNPTR is interoperable with any C function pointer type.
- Intrinsic Module Procedures

Named Constants

The ISO_C_BINDING named constants represent kind type parameters of data representations compatible with C types.

Intrinsic-Type Constants

The following table shows interoperable Fortran types and C Types.

Fortran Type	Named Constant for the KIND	C Type
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char, unsigned char
	C_SIZE_T	size_t

Fortran Type	Named Constant for the KIND	C Type
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
	C_INTMAX_T	intmax_t
	C_INTPTR_T	intptr_t
REAL	C_FLOAT	float
	C_DOUBLE	double
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex

Fortran Type	Named Constant for the KIND	C Type
	C_BOOL	_Bool
CHARACTER[1]	C_CHAR	char

[1]For character type, the length type parameter must be omitted or it must be specified by an initialization expression whose value is one.

For example, an integer type with the kind type parameter C_LONG is interoperable with the C integer type "long" or any C type derived from "long".

The value of C_INT will be a valid value for an integer kind type parameter on the processor. The values for the other integer named constants (C_INT*) will be a valid value for an integer kind type parameter on the processor, if any, or one of the following:

- -1 if the C processor defines the corresponding C type and there is no interoperating Fortran processor kind
- -2 if the C processor does not define the corresponding C type

The values of C_FLOAT, C_DOUBLE, and C_LONGDOUBLE will be a valid value for a real kind type parameter on the processor, if any, or one of the following:

- -1 if the C processor's type does not have a precision equal to the precision of any of the Fortran processor's real kinds
- -2 if the C processor's type does not have a range equal to the range of any of the Fortran processor's real kinds
- -3 if the C processor's type has neither the precision or range equal to the precision or range of any of the Fortran processor's real kinds
- -4 if there is no interoperating Fortran processor or kind for other reasons

The values of C_FLOAT_COMPLEX, C_DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX will be the same as those of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE, respectively.

The value of C_BOOL will be a valid value for a logical kind parameter on the processor, if any, or -1.

The value of C_CHAR is the character kind.

Character Constants

The following table shows interoperable named constants and C characters:

Fortran Named Constant	Definition	C Character
C_NULL_CHAR	null character	'\0'
C_ALERT	alert	'\a'
C_BACKSPACE	backspace	'\b'
C_FORM_FEED	form feed	'\f'
C_NEW_LINE	new line	'\n'
C_CARRIAGE_RETURN	carriage return	'\r'
C_HORIZONTAL_TAB	horizontal tab	'\t'
C_VERTICAL_TAB	vertical tab	'\v'

Derived-Type Constants

The constant C_NULL_PTR is of type C_PTR; it has the value of a C null data pointer. The constant C_NULL_FUNPTR is of type C_FUNPTR; it has the value of a C null function pointer.

Intrinsic Module Procedures

The following procedures are provided with the ISO_C_BINDING intrinsic module:

- C_ASSOCIATED
- C_F_POINTER
- C_F_PROCPOINTER
- C_FUNLOC
- C_LOC

None of the procedures are pure.

ISO_FORTRAN_ENV Module

The ISO_FORTRAN_ENV intrinsic module provides information about the Fortran run-time environment. It takes the following form:

```
USE, INTRINSIC :: ISO_FORTRAN_ENV
```

This intrinsic module provides the named constants you can use to get information on the Fortran environment. They are all scalars of type default integer.

Named Constant	Definition
CHARACTER_STORAGE_SIZE	Is the size of the character storage unit expressed in bits.
ERROR_UNIT	Identifies the preconnected external unit used for error reporting.
FILE_STORAGE_SIZE	Is the size of the file storage unit expressed in bits. To use this constant, compiler option assume byterecl must be enabled.
INPUT_UNIT	Identifies the preconnected external unit as the one specified by an asterisk in a READ statement. To use this constant, compiler option assume noold_unit_star must be enabled.
IOSTAT_END	Is the value assigned to the variable specified in an IOSTAT= specifier if an end-of-file condition occurs during execution of an input/output statement and no error condition occurs.
IOSTAT_EOR	Is the value assigned to the variable specified in an IOSTAT= specifier if an end-of-record condition occurs during execution of an input/output statement and no error condition occurs.
NUMERIC_STORAGE_SIZE	Is the size of the numeric storage unit expressed in bits.
OUTPUT_UNIT	Identifies the preconnected external unit as the one specified by an asterisk in a WRITE statement. To use this constant, compiler option assume noold_unit_star must be enabled.

IEEE Intrinsic Modules and Procedures

Intel Fortran includes IEEE intrinsic modules that support IEEE arithmetic and exception handling. The modules contain derived data types that include named constants for controlling the level of support, and intrinsic module procedures. The modules and procedures are Fortran 2003 features.

To include an IEEE module in your program, specify the intrinsic module name in a `USE` statement; for example:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
```

You must include the `INTRINSIC` attribute or the processor will look for a non-intrinsic module. Once you include a module, all related intrinsic procedures are defined.

There are three IEEE intrinsic modules described in this section:

- IEEE_ARITHMETIC
- IEEE_EXCEPTIONS
- IEEE_FEATURES

Determining Availability of IEEE Features

Before using a particular IEEE feature, you can determine whether your processor supports it by using the IEEE inquiry functions (listed in Table 1).

For example:

- To determine whether IEEE arithmetic is available for a particular kind of real, use intrinsic module function IEEE_SUPPORT_DATATYPE
- To determine whether you can change a rounding mode, use intrinsic module function IEEE_SUPPORT_ROUNDING.
- To determine whether a divide operation will be supported with the accuracy specified by the IEEE standard, use intrinsic module function IEEE_SUPPORT_DIVIDE.
- To determine whether you can control halting after an exception has occurred, use intrinsic module function IEEE_SUPPORT_HALTING.
- To determine which exceptions are supported in a scoping unit, use intrinsic module function IEEE_SUPPORT_FLAG.
- To determine whether all IEEE features are supported, use intrinsic module function IEEE_SUPPORT_STANDARD.

Restrictions for IEEE Intrinsic Procedures

The following intrinsic procedures can only be invoked if IEEE_SUPPORT_DATATYPE is true for their arguments:

IEEE_CLASS	IEEE_SUPPORT_DENORMAL
IEEE_COPY_SIGN	IEEE_SUPPORT_DIVIDE
IEEE_IS_FINITE	IEEE_SUPPORT_INR
IEEE_NEGATIVE	IEEE_SUPPORT_IO
IEEE_IS_NORMAL	IEEE_SUPPORT_NAN
IEEE_LOGB	IEEE_SUPPORT_ROUNDING
IEEE_NEXT_AFTER	IEEE_SUPPORT_SQRT
IEEE_REM	IEEE_SUPPORT_UNORDERED
IEEE_RINT	IEEE_SUPPORT_VALUE
IEEE_SCALB	IEEE_VALUE
IEEE_SET_ROUNDING_MODE[1]	IEEE_SUPPORT_ROUNDING

1: IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X) must also be true.

For example, the IEEE_IS_NORMAL(X) function can only be invoked if IEEE_SUPPORT_DATATYPE(X) has the value true. Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

...

  IF IEEE_SUPPORT_DATATYPE(X) THEN

   IF IEEE_IS_NORMAL(X) THEN

      PRINT *, ' X is a 'normal' '

   ELSE

      PRINT *, ' X is not 'normal' '

   ELSE

      PRINT *, ' X is not a supported IEEE type '

ENDIF

...
```

Certain other IEEE intrinsic module procedures have similar restrictions:

- IEEE_IS_NAN(X) can only be invoked if IEEE_SUPPORT_NAN(X) has the value true.
- IEEE_SET_HALTING_MODE(FLAG, HALTING) can only be invoked if IEEE_SUPPORT_HALTING(FLAG) has the value true.
- IEEE_GET_UNDERFLOW_MODE(GRADUAL) can only be invoked if IEEE_SUPPORT_UNDERFLOW_CONTROL(X) is true for some X.

For intrinsic module function IEEE_CLASS(X), some of the possible return values also have restrictions. These restrictions are also true for argument CLASS in intrinsic module function IEEE_VALUE(X, CLASS):

- IEEE_POSITIVE_INF and IEEE_NEGATIVE_INF can only be returned if IEEE_SUPPORT_INF(X) has the value true.
- IEEE_POSITIVE_DENORMAL and IEEE_NEGATIVE_DENORMAL can only be returned if IEEE_SUPPORT_DENORMAL(X) has the value true.
- IEEE_SIGNALING_NAN and IEEE_QUIET_NAN can only be returned if IEEE_SUPPORT_NAN(X) has the value true.

IEEE_ARITHMETIC Intrinsic Module

The IEEE_ARITHMETIC module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures. This module and its procedures are Fortran 2003 features.

The derived types in the intrinsic modules have components that are private. The IEEE_ARITHMETIC intrinsic module supports IEEE arithmetic and features. It defines the following derived types:

- *IEEE_CLASS_TYPE:* Identifies a class of floating-point values. Its values are the following named constants:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUIET_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO
IEEE_OTHER_VALUE	

- *IEEE_ROUND_TYPE:* Identifies a rounding mode. Its values are the following named constants:

IEEE_NEAREST	IEEE_TO_ZERO
IEEE_UP	IEEE_OTHER[1]
IEEE_DOWN	

1: Specifies the rounding mode does not conform to the IEEE standard.

The IEEE_ARITHMETIC intrinsic module also defines the following operators:

- Elemental operator = = for two values of one of the above types to return true if the values are the same; otherwise, false.
- Elemental operator /= for two values of one of the above types to return true if the values differ; otherwise, false.

The IEEE_ARITHMETIC module includes support for IEEE_EXCEPTIONS module, and public entities in IEEE_EXCEPTIONS module are also public in the IEEE_ARITHMETIC module.

IEEE_EXCEPTIONS Intrinsic Module

The IEEE_EXCEPTIONS module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures. This module and its procedures are Fortran 2003 features.

The derived types in the intrinsic modules have components that are private. The IEEE_EXCEPTIONS intrinsic module supports the setting, clearing, saving, restoring, or testing of exception flags. It defines the following derived types:

- *IEEE_FLAG_TYPE:* Identifies an exception flag for errors that occur during an IEEE arithmetic operation or assignment. Its values are the following named constants:

IEEE_INVALID	IEEE_DIVIDE_BY_ZERO
IEEE_OVERFLOW	IEEE_INEXACT
IEEE_UNDERFLOW	

  Each of the above exceptions has a flag whose value is either quiet or signaling. The initial value is quiet and it signals when the associated exception occurs. To determine the value of a flag, use intrinsic module subroutine IEEE_GET_FLAG. To change the status for a flag, use intrinsic module subroutine IEEE_SET_FLAG or IEEE_SET_STATUS.

  If a flag is signaling on entry to a procedure, the processor sets it to quiet on entry and restores it to signaling on return.

  If a flag is quiet on entry to a procedure with access to modules IEEE_ARITHMETIC or IEEE_EXCEPTIONS, and is signaling on return, the processor will not restore it to quiet.

  The IEEE_FLAG_TYPE module also defines the following named array constants:

  - IEEE_USUAL=(/IEEE_OVERFLOW,IEEE_DIVIDE_BY_ZERO, IEEE_INVALID/)
  - IEEE_ALL=(/IEEE_USUAL,IEEE_UNDERFLOW,IEEE_INEXACT/)

- *IEEE_STATUS_TYPE:* Saves the current floating-point status.

The IEEE_ARITHMETIC module includes support for IEEE_EXCEPTIONS module, and public entities in IEEE_EXCEPTIONS module are also public in the IEEE_ARITHMETIC module.

IEEE_FEATURES Intrinsic Module

The IEEE_FEATURES module contains derived data types that include named constants for controlling the level of support, and intrinsic module procedures. This module and its procedures are Fortran 2003 features.

The derived types in the intrinsic modules have components that are private. The IEEE_FEATURES intrinsic module supports specification of essential IEEE features. It defines the following derived types:

- *IEEE_FEATURES_TYPE:* Specifies IEEE features. Its values are the following named constants:

IEEE_DATATYPE	IEEE_INF

IEEE_DIVIDE	IEEE_NAN
IEEE_ROUNDING	IEEE_INEXACT_FLAG
IEEE_SQRT	IEEE_INVALID_FLAG
IEEE_DENORMAL	IEEE_UNDERFLOW_FLAG
IEEE_HALTING	

- *IEEE_STATUS_TYPE:* Saves the current floating-point status.

IEEE Intrinsic Modules Quick Reference Tables

This topic contains quick reference tables for categories of IEEE intrinsic modules.

Table 588: Categories of Intrinsic Module Functions

Category	Sub-category	Description
IEEE	Arithmetic	Test IEEE values or provide features: IEEE_CLASS, IEEE_COPY_SIGN, IEEE_IS_FINITE, IEEE_IS_NAN, IEEE_IS_NORMAL, IEEE_IS_NEGATIVE, IEEE_LOGB, IEEE_NEXT_AFTER, IEEE_REM, IEEE_RINT, IEEE_SCALB, IEEE_UNORDERED, IEEE_VALUE
	Inquiry	Returns whether the processor supports certain exceptions or IEEE features: IEEE_SUPPORT_DATATYPE, IEEE_SUPPORT_DENORMAL, IEEE_SUPPORT_DIVIDE, IEEE_SUPPORT_FLAG, IEEE_SUPPORT_HALTING, IEEE_SUPPORT_INF, IEEE_SUPPORT_IO, IEEE_SUPPORT_NAN, IEEE_SUPPORT_ROUNDING, IEEE_SUPPORT_SQRT, IEEE_SUPPORT_STANDARD, IEEE_SUPPORT_UNDERFLOW_CONTROL
	Transformational	Returns the kind type parameter of an IEEE value: IEEE_SELECTED_REAL_KIND

Table 589: Summary of Generic Module Intrinsic Functions

Generic Function	Class	Value Returned
IEEE_CLASS (X)	E	The IEEE class
IEEE_COPY_SIGN (X, Y)	E	An argument with a copied sign; the IEEE copysign function
IEEE_IS_FINITE (X)	E	Whether a value is finite
IEEE_IS_NAN (X)	E	Whether a value is NaN
IEEE_IS_NEGATIVE (X)	E	Whether a value is negative
IEEE_IS_NORMAL (X)	E	Whether a value is normal
IEEE_LOGB (X)	E	An exponent in IEEE floating-point format; the IEEE logb function
IEEE_NEXT_AFTER (X, Y)	E	The next representable value after X toward Y; the IEEE nextafter function
IEEE_REM (X, Y)	E	The result of a remainder operation; the IEEE rem function
IEEE_RINT (X)	E	An integer value rounded according to the current rounding mode
IEEE_SCALB (X, I)	E	The value of X multiplied by 2**I; the IEEE scalb function
IEEE_SELECTED_REAL_KIND ([P] [, R])	T	The kind type parameter for an IEEE real
IEEE_SUPPORT_DATATYPE ([X])	I	Whether IEEE arithmetic is supported
IEEE_SUPPORT_DENORMAL ([X])	I	Whether denormalized numbers are supported
IEEE_SUPPORT_DIVIDE ([X])	I	Whether divide accuracy compares to IEEE standard
IEEE_SUPPORT_FLAG (FLAG [, X])	I	Whether an exception is supported
IEEE_SUPPORT_HALTING (FLAG)	I	Whether halting after and exception is supported

Generic Function	Class	Value Returned
IEEE_SUPPORT_INF ([X])	I	Whether IEEE infinities are supported
IEEE_SUPPORT_IO ([X])	I	Whether IEEE base conversion rounding is supported during formatted I/O
IEEE_SUPPORT_NAN ([X])	I	Whether IEEE Not-A-Number is supported
IEEE_SUPPORT_ROUNDING (ROUND_VALUE [, X])	I	Whether a particular rounding mode is supported
IEEE_SUPPORT_SQRT ([X])	I	Whether IEEE square root is supported
IEEE_SUPPORT_STANDARD ([X])	I	Whether all IEEE capabilities are supported
IEEE_SUPPORT_UNDERFLOW_CONTROL(X)	I	Whether control of underflow mode is supported
IEEE_UNORDERED (X, Y)	E	Whether one or both arguments are NaN; the IEEE unordered function
IEEE_VALUE (X, CLASS)	E	An IEEE value

Table 590: Intrinsic Modules Subroutines

Subroutine	Value Returned or Result
IEEE_GET_FLAG (FLAG, FLAG_VALUE)[1]	Whether an exception flag is signalling
IEEE_GET_HALTING_MODE (FLAG, HALTING)[1]	The current halting mode for an exception
IEEE_GET_ROUNDING_MODE (ROUND_VALUE)	The current IEEE rounding mode
IEEE_GET_STATUS (STATUS_VALUE)	The current state of the floating-point environment
IEEE_GET_UNDERFLOW_MODE (GRADUAL)	The current underflow mode
IEEE_SET_FLAG (FLAG, FLAG_VALUE)[1]	Assigns a value to an exception flag
IEEE_SET_HALTING_MODE (FLAG, HALTING)[1]	Controls the halting mode after an exception

Subroutine	Value Returned or Result
IEEE_SET_ROUNDING_MODE (ROUND_VALUE)	Sets the IEEE rounding mode
IEEE_SET_STATUS (STATUS_VALUE)	Restores the state of the floating-point environment

1 : An elemental subroutine

Block Data Program Units Overview

A block data program unit provides initial values for nonpointer variables in named common blocks. For more information, see BLOCK DATA.

Examples

An example of a block data program unit follows:

```
BLOCK DATA WORK

   COMMON /WRKCOM/ A, B, C (10,10)

   DATA A /1.0/, B /2.0/, C /100*0.0/

END BLOCK DATA WORK
```

Functions, Subroutines, and Statement Functions

Functions, subroutines, and statement functions are user-written subprograms that perform computing procedures. The computing procedure can be either a series of arithmetic operations or a series of Fortran statements. A single subprogram can perform a computing procedure in several places in a program, to avoid duplicating a series of operations or statements in each place.

The following table shows the statements that define these subprograms, and how control is transferred to the subprogram:

Subprogram	Defining Statements	Control Transfer Method
Function	FUNCTION or ENTRY	Function reference[1]
Subroutine	SUBROUTINE or ENTRY	CALL statement[2]
Statement function	Statement function definition	Function reference

Subprogram	Defining Statements	Control Transfer Method

[1] A function can also be invoked by a defined operation (see Defining Generic Operators).

[2] A subroutine can also be invoked by a defined assignment (see Defining Generic Assignment).

A *function reference* is used in an expression to invoke a function; it consists of the function name and its actual arguments. The function reference returns a value to the calling expression that is used to evaluate the expression.

See Also

- Program Units and Procedures
- General Rules for Function and Subroutine Subprograms
- Functions Overview
- Subroutines Overview
- Statement Functions Overview
- General rules for function and subroutine subprograms
- Functions
- Subroutines
- Statement functions
- ENTRY statement
- CALL statement

General Rules for Function and Subroutine Subprograms

A subprogram can be an external, module, or internal subprogram. The END statement for an internal or module subprogram must be END SUBROUTINE [name] for a subroutine, or END FUNCTION [name] for a function. In an external subprogram, the SUBROUTINE and FUNCTION keywords are optional.

If a subprogram name appears after the END statement, it must be the same as the name specified in the SUBROUTINE or FUNCTION statement.

Function and subroutine subprograms can change the values of their arguments, and the calling program can use the changed values.

A SUBROUTINE or FUNCTION statement can be optionally preceded by an OPTIONS statement.

Dummy arguments (except for dummy pointers or dummy procedures) can be specified with an intent and can be made optional.

Recursive Procedures

A recursive procedure is a function or subroutine that references itself, either directly or indirectly. For more information, see RECURSIVE.

Pure Procedures

A pure procedure is a user-defined procedure that has no side effects. Pure procedures are a feature of Fortran 95. For more information, see PURE.

Elemental Procedures

An elemental procedure is a user-defined procedure that is a restricted form of pure procedure. For more information, see PURE and ELEMENTAL.

Functions Overview

A *function* subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression. For more information, see FUNCTION.

See Also

- Functions, Subroutines, and Statement Functions
- RESULT Keyword Overview
- Function References
- RESULT Keyword
- Function References

RESULT Keyword Overview

If you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. For more information, see RESULT.

Function References

Functions are invoked by a function reference in an expression or by a defined operation.

A function reference takes the following form:

fun ([*a-arg* [, *a-arg*] ...])

fun	Is the name of the function subprogram.
a-arg	Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the function. The keyword is assigned a value when the procedure is invoked. Each actual argument must be a variable, an expression, or the name of a procedure. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

Description

When a function is referenced, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

Execution of the function produces a result that is assigned to the function name or to the result name, depending on whether the RESULT keyword was specified.

The program unit uses the result value to complete the evaluation of the expression containing the function reference.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

If a dummy argument is specified with the INTENT attribute, its use may be limited. A dummy argument whose intent is not specified is subject to the limitations of its associated actual argument.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Functions Not Allowed as Actual Arguments).

Examples

Consider the following example:

```
X = 2.0

NEW_COS = COS(X)        ! A function reference
```

Intrinsic function COS calculates the cosine of 2.0. The value -0.4161468 is returned (in place of COS(X)) and assigned to NEW_COS.

See Also

- Functions Overview
- INTENT attribute
- Defining Generic Operators
- Dummy Procedure Arguments
- Intrinsic Procedures
- Optional arguments
- RESULT keyword
- FUNCTION statement

- Argument Association

Subroutines Overview

A *subroutine* subprogram is invoked in a CALL statement or by a defined assignment statement, and does not return a particular value. For more information, see SUBROUTINE.

Statement Functions Overview

A statement function is a procedure defined by a single statement in the same program unit in which the procedure is referenced. For more information, see Statement Function.

External Procedures

External procedures are user-written functions or subroutines. They are located outside of the main program and can't be part of any other program unit.

External procedures can be invoked by the main program or any procedure of an executable program.

In Fortran 95/90, external procedures can include internal subprograms (defining internal procedures). Internal subprograms are placed after a CONTAINS statement.

An external procedure can reference itself (directly or indirectly).

The interface of an external procedure is implicit unless an interface block is supplied for the procedure.

See Also

- Program Units and Procedures
- Functions, Subroutines, and Statement Functions
- Procedure Interfaces

Building Applications for details on passing arguments

Internal Procedures

Internal procedures are functions or subroutines that follow a CONTAINS statement in a program unit. The program unit in which the internal procedure appears is called its *host*.

Internal procedures can appear in the main program, in an external subprogram, or in a module subprogram.

An internal procedure takes the following form:

CONTAINS

internal-subprogram

[*internal-subprogram*] ...

internal-subprogram Is a function or subroutine subprogram that defines the procedure. An internal subprogram must not contain any other internal subprograms.

Description

Internal procedures are the same as external procedures, except for the following:

- Only the host program unit can use an internal procedure.
- An internal procedure has access to host entities by host association; that is, names declared in the host program unit are useable within the internal procedure.
- In Fortran 95/90, the name of an internal procedure must not be passed as an argument to another procedure. However, Intel® Fortran allows an internal procedure name to be passed as an actual argument to another procedure.
- An internal procedure must not contain an ENTRY statement.

An internal procedure can reference itself (directly or indirectly); it can be referenced in the execution part of its host and in the execution part of any internal procedure contained in the same host (including itself).

The interface of an internal procedure is always explicit.

Examples

The following example shows an internal procedure:

```
PROGRAM COLOR_GUIDE

...

 CONTAINS

  FUNCTION HUE(BLUE)   ! An internal procedure

  ...

  END FUNCTION HUE

END PROGRAM
```

The following example program contains an internal subroutine `find`, which performs calculations that the main program then prints. The variables `a`, `b`, and `c` declared in the host program are also known to the internal subroutine.

```
program INTERNAL
! shows use of internal subroutine and CONTAINS statement
   real a,b,c
   call find
   print *, c
contains
   subroutine find
    read *, a,b
    c = sqrt(a**2 + b**2)
   end subroutine find
end
```

See Also

- Program Units and Procedures
- Functions, Subroutines, and Statement Functions
- Host association
- Procedure Interfaces
- CONTAINS

Argument Association

Procedure arguments provide a way for different program units to access the same data.

When a procedure is referenced in an executable program, the program unit invoking the procedure can use one or more *actual* arguments to pass values to the procedure's *dummy* arguments. The dummy arguments are associated with their corresponding actual arguments when control passes to the subprogram.

In general, when control is returned to the calling program unit, the last value assigned to a dummy argument is assigned to the corresponding actual argument.

An actual argument can be a variable, expression, or procedure name. The type and kind parameters, and rank of the actual argument must match those of its associated dummy argument.

A dummy argument is either a dummy data object, a dummy procedure, or an alternate return specifier (*). Except for alternate return specifiers, dummy arguments can be optional.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments.

A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

A scalar dummy argument can be associated with only a scalar actual argument.

If a dummy argument is an array, it must be no larger than the array that is the actual argument. You can use adjustable arrays to process arrays of different sizes in a single subprogram.

An actual argument associated with a dummy argument that is allocatable or a pointer must have the same type parameters as the dummy argument.

A dummy argument referenced as a subprogram must be associated with an actual argument that has been declared EXTERNAL or INTRINSIC in the calling routine.

If a scalar dummy argument is of type character, its length must not be greater than the length of its associated actual argument.

If the character dummy argument's length is specified as *(*) (assumed length), it uses the length of the associated actual argument.

Once an actual argument has been associated with a dummy argument, no action can be taken that affects the value or availability of the actual argument, except indirectly through the dummy argument. For example, if the following statement is specified:

```
CALL SUB_A (B(2:6), B(4:10))
```

B(4:6) must not be defined, redefined, or become undefined through either dummy argument, since it is associated with both arguments. However, B(2:3) is definable through the first argument, and B(7:10) is definable through the second argument.

Similarly, if any part of the actual argument is defined through a dummy argument, the actual argument can only be referenced through that dummy argument during execution of the procedure. For example, if the following statements are specified:

```
MODULE MOD_A
  REAL :: A, B, C, D
END MODULE MOD_A
PROGRAM TEST
  USE MOD_A
  CALL SUB_1 (B)
  ...
END PROGRAM TEST
SUBROUTINE SUB_1 (F)
  USE MOD_A
  ...
  WRITE (*,*) F
END SUBROUTINE SUB_1
```

Variable B must not be directly referenced during the execution of SUB_1 because it is being defined through dummy argument F. However, B can be indirectly referenced through F (and directly referenced when SUB_1 completes execution).

The following sections provide more details on arguments:

- Optional arguments
- The different kinds of arguments:
  - Array arguments
  - Pointer arguments
  - Assumed-length character arguments
  - Character constant and Hollerith arguments
  - Alternate return arguments
  - Dummy procedure arguments
- References to generic procedures
- References to non-Fortran procedures (%REF, %VAL, and %LOC)

Optional Arguments

Dummy arguments can be made optional if they are declared with the OPTIONAL attribute. In this case, an actual argument does not have to be supplied for it in a procedure reference.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments. A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

Positional arguments (if any) must appear first in an actual argument list, followed by keyword arguments (if any). If an optional argument is the last positional argument, it can simply be omitted if desired.

However, if the optional argument is to be omitted but it is not the last positional argument, keyword arguments must be used for any subsequent arguments in the list.

Optional arguments must have explicit procedure interfaces so that appropriate argument associations can be made.

The PRESENT intrinsic function can be used to determine if an actual argument is associated with an optional dummy argument in a particular reference.

The following example shows optional arguments:

```
PROGRAM RESULT

TEST_RESULT = LGFUNC(A, B=D)

...

CONTAINS

  FUNCTION LGFUNC(G, H, B)

    OPTIONAL H, B

    ...

  END FUNCTION

END
```

In the function reference, A is a positional argument associated with required dummy argument G. The second actual argument D is associated with optional dummy argument B by its keyword name (B). No actual argument is associated with optional argument H.

The following shows another example:

```
! Arguments can be passed out of order, but must be

!  associated with the correct dummy argument.

CALL EXT1 (Z=C, X=A, Y=B)

. . .

END

SUBROUTINE EXT1(X,Y,Z)

   REAL X, Y

   REAL, OPTIONAL :: Z

   . . .

END SUBROUTINE
```

In this case, argument A is associated with dummy argument X by explicit assignment. Once EXT1 executes and returns, A is no longer associated with X, B is no longer associated with Y, and C is no longer associated with Z.

See Also

- Argument Association
- OPTIONAL attribute
- PRESENT intrinsic function
- Argument association
- CALL
- Function References

Array Arguments

Arrays are sequences of elements. Each element of an actual array is associated with the element of the dummy array that has the same position in array element order.

If the dummy argument is an explicit-shape or assumed-size array, the size of the dummy argument array must not exceed the size of the actual argument array.

The type and kind parameters of an explicit-shape or assumed-size dummy argument must match the type and kind parameters of the actual argument, but their ranks need not match.

If the dummy argument is an assumed-shape array, the size of the dummy argument array is equal to the size of the actual argument array. The associated actual argument must not be an assumed-size array or a scalar (including a designator for an array element or an array element substring).

If the actual argument is an array section with a vector subscript, the associated dummy argument must not be defined.

The declaration of an array used as a dummy argument can specify the lower bound of the array.

If a dummy argument is allocatable, the actual argument must be allocatable and the type parameters and ranks must agree. An example of an allocatable function with allocatable arrays appears in FUNCTION.

Dummy argument arrays declared as assumed-shape, deferred-shape, or pointer arrays require an explicit interface visible to the caller.

See Also

- Argument Association
- Arrays
- Array association
- Argument association
- Array Elements
- Explicit-Shape Specifications
- Assumed-Shape Specifications
- Assumed-Size Specifications

Pointer Arguments

An argument is a pointer if it is declared with the POINTER attribute.

When a procedure is invoked, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target.

The pointer association status of the dummy argument can change during the execution of the procedure, and any such changes are reflected in the actual argument.

If both the dummy and actual arguments are pointers, an explicit interface is required.

A dummy argument that is a pointer can be associated only with an actual argument that is a pointer. However, an actual argument that is a pointer can be associated with a nonpointer dummy argument. In this case, the actual argument is associated with a target and the dummy argument, through argument association, also becomes associated with that target.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument when the procedure is invoked.

If the dummy argument has the TARGET attribute, and is either a scalar or assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, the following occurs:

- Any pointer associated with the actual argument becomes associated with the corresponding dummy argument when the procedure is invoked.
- Any pointers associated with the dummy argument remain associated with the actual argument when execution of the procedure completes.

If the dummy argument has the TARGET attribute, and is an explicit-shape or assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript, association of actual and corresponding dummy arguments when the procedure is invoked or when execution is completed is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have that attribute or is an array section with a vector subscript, any pointer associated with the dummy argument becomes undefined when execution of the procedure completes.

See Also

- Argument Association
- POINTER statement and attribute
- Pointer assignments
- TARGET statement and attribute
- Argument association

Assumed-Length Character Arguments

An assumed-length character argument is a dummy argument that assumes the length attribute of its corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.

A character array dummy argument can also have an assumed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The assumed length and the array declarator together determine the size of the assumed-length character array.

The following example shows an assumed-length character argument:

```
INTEGER FUNCTION ICMAX(CVAR)
  CHARACTER*(*) CVAR
  ICMAX = 1
  DO I=2,LEN(CVAR)
    IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX=I
  END DO
  RETURN
END
```

The function ICMAX finds the position of the character with the highest ASCII code value. It uses the length of the assumed-length character argument to control the iteration. Intrinsic function LEN determines the length of the argument.

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression. Each of the following function references specifies a different length for the dummy argument:

```
CHARACTER VAR*10, CARRAY(3,5)*20
...
I1 = ICMAX(VAR)
I2 = ICMAX(CARRAY(2,2))
I3 = ICMAX(VAR(3:8))
I4 = ICMAX(CARRAY(1,3)(5:15))
I5 = ICMAX(VAR(3:4)//CARRAY(3,5))
```

See Also

- Argument Association
- LEN intrinsic function
- Argument association

Character Constant and Hollerith Arguments

If an actual argument is a character constant (for example, `'ABCD'`), the corresponding dummy argument must be of type character. If an actual argument is a Hollerith constant (for example, 4HABCD), the corresponding dummy argument must have a numeric data type.

The following example shows character and Hollerith constants being used as actual arguments:

```
SUBROUTINE S(CHARSUB, HOLLSUB, A, B)

EXTERNAL CHARSUB, HOLLSUB

...

CALL CHARSUB(A, 'STRING')

CALL HOLLSUB(B, 6HSTRING)
```

The subroutines CHARSUB and HOLLSUB are themselves dummy arguments of the subroutine S. Therefore, the actual argument `'STRING'` in the call to CHARSUB must correspond to a character dummy argument, and the actual argument 6HSTRING in the call to HOLLSUB must correspond to a numeric dummy argument.

See Also

- Argument Association
- Argument association

Alternate Return Arguments

Alternate return (dummy) arguments can appear in a subroutine argument list. They cause execution to transfer to a labeled statement rather than to the statement immediately following the statement that called the routine. The alternate return is indicated by an asterisk (*). (An alternate return is an obsolescent feature in Fortran 90 and Fortran 95.)

There can be any number of alternate returns in a subroutine argument list, and they can be in any position in the list.

An actual argument associated with an alternate return dummy argument is called an alternate return specifier; it is indicated by an asterisk (*) or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement.

Alternate returns cannot be declared optional.

In Fortran 90, you can also use the RETURN statement to specify alternate returns.

The following example shows alternate return actual and dummy arguments:

```
CALL MINN(X, Y, *300, *250, Z)

....

SUBROUTINE MINN(A, B, *, *, C)
```

See Also

- Argument Association
- Argument association

- SUBROUTINE
- CALL
- RETURN
- Obsolescent and Deleted Language Features

Dummy Procedure Arguments

If an actual argument is a procedure, its corresponding dummy argument is a dummy procedure. Dummy procedures can appear in function or subroutine subprograms.

The actual argument must be the specific name of an external, module, intrinsic, or another dummy procedure. If the specific name is also a generic name, only the specific name is associated with the dummy argument. Not all specific intrinsic procedures can appear as actual arguments. (For more information, see table Intrinsic Functions Not Allowed as Actual Arguments.)

The actual argument and corresponding dummy procedure must both be subroutines or both be functions.

If the interface of the dummy procedure is explicit, the type and kind parameters, and rank of the associated actual procedure must be the same as that of the dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a subroutine, the actual argument must be a subroutine or a dummy procedure.

If the interface of the dummy procedure is implicit and the procedure is referenced as a function or is explicitly typed, the actual argument must be a function or a dummy procedure.

Dummy procedures can be declared optional, but they must not be declared with an intent.

The following is an example of a procedure used as an argument:

```
REAL FUNCTION LGFUNC(BAR)
  INTERFACE
    REAL FUNCTION BAR(Y)
      REAL, INTENT(IN) :: Y
    END
  END INTERFACE
  ...
  LGFUNC = BAR(2.0)
  ...
END FUNCTION LGFUNC
```

See Also

- Argument Association
- Argument association

References to Generic Procedures

Generic procedures are procedures with different specific names that can be accessed under one generic (common) name. In FORTRAN 77, generic procedures were limited to intrinsic procedures. In the current Fortran standard, you can use generic interface blocks to specify generic properties for intrinsic and user-defined procedures.

If you refer to a procedure by using its generic name, the selection of the specific routine is based on the number of arguments and the type and kind parameters, and rank of each argument.

All procedures given the same generic name must be subroutines, or all must be functions. Any two must differ enough so that any invocation of the procedure is unambiguous.

The following sections describe references to generic intrinsic functions and show an example of using intrinsic function names.

References to Generic Intrinsic Functions

The generic intrinsic function name COS lists six specific intrinsic functions that calculate cosines: COS, DCOS, QCOS, CCOS, CDCOS, and CQCOS. These functions return different values: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), and COMPLEX(16) respectively.

If you invoke the cosine function by using the generic name COS, the compiler selects the appropriate routine based on the arguments that you specify. For example, if the argument is REAL(4), COS is selected; if it is REAL(8), DCOS is selected; and if it is COMPLEX(4), CCOS is selected.

You can also explicitly refer to a particular routine. For example, you can invoke the double-precision cosine function by specifying DCOS.

Procedure selection occurs independently for each generic reference, so you can use a generic reference repeatedly in the same program unit to access different intrinsic procedures.

You cannot use generic function names to select intrinsic procedures if you use them as follows:

- The name of a statement function
- A dummy argument name, a common block name, or a variable or array name

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Not all specific intrinsic functions can appear as actual arguments. (For more information, see Intrinsic Functions Not Allowed as Actual Arguments.)

A reference to a generic intrinsic procedure name in a program unit does not prevent use of the name for other purposes elsewhere in the program.

Normally, an intrinsic procedure name refers to the Fortran library procedure with that name. However, the name can refer to a user-defined procedure when the name appears in an EXTERNAL statement.

NOTE. If you call an intrinsic procedure by using the wrong number of arguments or an incorrect argument type, the compiler assumes you are referring to an external procedure. For example, intrinsic procedure SIN requires one argument; if you specify two arguments, such as SIN(10,4), the compiler assumes SIN is external and not intrinsic.

The data type of an intrinsic procedure does not change if you use an IMPLICIT statement to change the implied data type rules.

Intrinsic and user-defined procedures cannot have the same name if they appear in the same program unit.

Examples

The following example shows the local and global properties of an intrinsic function name. It uses the name SIN in different procedures as follows:

- The name of a statement function
- The generic name of an intrinsic function
- The specific name of an intrinsic function
- The name of a user-defined function

Using and Redefining an Intrinsic Function Name

```
!    Compare ways of computing sine
     PROGRAM SINES
       DOUBLE PRECISION X, PI
       PARAMETER (PI=3.14159265358979323 8D0)
       COMMON V(3)
!    Define SIN as a statement function  1
        SIN(X) = COS(PI/2-X)
        print *
        print *, "                    Way of computing SIN(X)"
        print *
        print *, "      X       Statement   Intrinsic   Intrinsic     User's "
        print *, "              function      DSIN       SIN as arg     SIN  "
        print *
        DO X = -PI, PI, PI/2
                    CALL COMPUT(X)
!    References the statement function SIN  2
                    WRITE (6,100) X, SIN(X), V
         END DO
100      FORMAT (5F12.7)
       END
       SUBROUTINE COMPUT(Y)
         DOUBLE PRECISION Y
!      Use intrinsic function SIN  - double-precision DSIN will be passed as an actual argument
  3
         INTRINSIC SIN
         COMMON V(3)
```

```
!       Makes the generic name SIN reference the double-precision sine DSIN  4
          V(1) = SIN(Y)
!       Use intrinsic function SIN as an actual argument - will pass DSIN  5
           CALL SUB(REAL(Y),SIN)
        END
        SUBROUTINE SUB(A,S)
!       Declare SIN as name of a user function  6
            EXTERNAL SIN
!       Declare SIN as type DOUBLE PRECISION  7
            DOUBLE PRECISION SIN
            COMMON V(3)
!       Evaluate intrinsic function SIN passed as the dummy argument  8
            V(2) = S(A)
!       Evaluate user-defined SIN function  9
            V(3) = SIN(A)
        END
!       Define the user SIN function  10
        DOUBLE PRECISION FUNCTION SIN(X)
           INTEGER FACTOR
           SIN = X - X**3/FACTOR(3) + X**5/FACTOR(5)     &
               - X**7/FACTOR(7)
        END
!       Compute the factorial of N
        INTEGER FUNCTION FACTOR(N)
           FACTOR = 1
           DO I=N,1,-1
```

```
            FACTOR = FACTOR * I
          END DO
        END
```

1 The statement function named SIN is defined in terms of the generic function name COS. Because the argument of COS is double precision, the double-precision cosine function is evaluated. The statement function SIN is itself single precision.

2 The statement function SIN is called.

3 The name SIN is declared intrinsic so that the single-precision intrinsic sine function can be passed as an actual argument at 5.

4 The generic function name SIN is used to refer to the double-precision sine function.

5 The single-precision intrinsic sine function is used as an actual argument.

6 The name SIN is declared a user-defined function name.

7 The type of SIN is declared double precision.

8 The single-precision sine function passed at 5 is evaluated.

9 The user-defined SIN function is evaluated.

10 The user-defined SIN function is defined as a simple Taylor series using a user-defined function FACTOR to compute the factorial function.

See Also

- References to Generic Procedures
- EXTERNAL attribute
- INTRINSIC attribute
- Intrinsic procedures
- Names

References to Elemental Intrinsic Procedures

An *elemental intrinsic procedure* has scalar dummy arguments that can be called with scalar or array actual arguments. If actual arguments are array-valued, they must have the same shape. There are many elemental intrinsic functions, but only one elemental intrinsic subroutine (MVBITS).

If the actual arguments are scalar, the result is scalar. If the actual arguments are array-valued, the scalar-valued procedure is applied element-by-element to the actual argument, resulting in an array that has the same shape as the actual argument.

The values of the elements of the resulting array are the same as if the scalar-valued procedure had been applied separately to the corresponding elements of each argument.

For example, if A and B are arrays of shape (5,6), MAX(A, 0.0, B) is an array expression of shape (5,6) whose elements have the value MAX(A (i, j), 0.0, B (i, j)), where *i* = 1, 2,..., 5, and *j* = 1, 2,..., 6.

A reference to an elemental intrinsic procedure is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

Examples

Consider the following:

```
REAL, DIMENSION (2) :: a, b

a(1) = 4; a(2) = 9

b = SQRT(a)                ! sets b(1) = SQRT(a(1)), and b(2) = SQRT(a(2))
```

See Also

- References to Generic Procedures
- Arrays
- Intrinsic Procedures

References to Non-Fortran Procedures

When a procedure is called, Fortran (by default) passes the address of the actual argument, and its length if it is of type character. To call non-Fortran procedures, you may need to pass the actual arguments in a form different from that used by Fortran.

The built-in functions %REF and %VAL let you change the form of an actual argument. You must specify these functions in the actual argument list of a CALL statement or function reference. You cannot use them in any other context.

%LOC computes the internal address of a storage item.

Procedure Interfaces

Every procedure has an interface, which consists of the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units.

If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration). The following table shows which procedures have implicit or explicit interfaces:

Kind of Procedure	Interface
External procedure	Implicit [1]
Module procedure	Explicit
Internal procedure	Explicit
Intrinsic procedure	Explicit
Dummy procedure	Implicit [1]
Statement function	Implicit

[1] Unless an interface block is supplied for the procedure.

The interface of a recursive subroutine or function is explicit within the subprogram that defines it.

An explicit interface can appear in a procedure's definition, in an interface block, or both. (Internal procedures must not appear in an interface block.)

The following sections describe when explicit interfaces are required, how to define explicit interfaces, and how to define generic names, operators, and assignment.

Examples

An example of an interface block follows:

```
INTERFACE
  SUBROUTINE Ext1 (x, y, z)
  REAL, DIMENSION (100,100) :: x, y, z
  END SUBROUTINE Ext1
  SUBROUTINE Ext2 (x, z)
  REAL x
  COMPLEX (KIND = 4) z (2000)
  END SUBROUTINE Ext2
  FUNCTION Ext3 (p, q)
  LOGICAL Ext3
  INTEGER p (1000)
  LOGICAL q (1000)
  END FUNCTION Ext3
END INTERFACE
```

Determining When Procedures Require Explicit Interfaces

A procedure must have an explicit interface in the following cases:

- If the procedure has any of the following:
  - A dummy argument that has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET, VALUE, or VOLATILE attribute
  - A dummy argument that is an assumed-shape array
  - A result that is an array, or a pointer, or is allocatable (functions only)
  - A result whose length is neither assumed nor a constant (character functions only)
- If a reference to the procedure appears as follows:
  - With an argument keyword
  - As a reference by its generic name
  - As a defined assignment (subroutines only)

  - In an expression as a defined operator (functions only)
  - In a context that requires it to be pure
- If the procedure is elemental

See Also

- Procedure Interfaces
- Optional arguments
- Array arguments
- Pointer arguments
- CALL
- Function references
- Pure procedures
- Elemental procedures
- Defining Generic Names for Procedures
- Defining Generic Operators
- Defining Generic Assignment

Defining Explicit Interfaces

Interface blocks define explicit interfaces for external or dummy procedures. They can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines.

See Also

- Procedure Interfaces
- INTERFACE

Defining Generic Names for Procedures

An interface block can be used to specify a generic name to reference all of the procedures within the interface block.

The initial line for such an interface block takes the following form:

INTERFACE *generic-name*

generic-name	Is the generic name. It can be the same as any of the procedure names in the interface block, or the same as any accessible generic name (including a generic intrinsic name).

This kind of interface block can be used to extend or redefine a generic intrinsic procedure.

The procedures that are given the generic name must be the same kind of subprogram: all must be functions, or all must be subroutines.

Any procedure reference involving a generic procedure name must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

The following is an example of a procedure interface block defining a generic name:

```
INTERFACE GROUP_SUBS

  SUBROUTINE INTEGER_SUB (A, B)

    INTEGER, INTENT(INOUT) :: A, B

  END SUBROUTINE INTEGER_SUB

  SUBROUTINE REAL_SUB (A, B)

    REAL, INTENT(INOUT) :: A, B

  END SUBROUTINE REAL_SUB

  SUBROUTINE COMPLEX_SUB (A, B)

    COMPLEX, INTENT(INOUT) :: A, B

  END SUBROUTINE COMPLEX_SUB

END INTERFACE
```

The three subroutines can be referenced by their individual specific names or by the group name GROUP_SUBS.

The following example shows a reference to INTEGER_SUB:

```
INTEGER V1, V2

CALL GROUP_SUBS (V1, V2)
```

Consider the following:

```
INTERFACE LINE_EQUATION
  SUBROUTINE REAL_LINE_EQ(X1,Y1,X2,Y2,M,B)
    REAL,INTENT(IN) :: X1,Y1,X2,Y2
    REAL,INTENT(OUT) :: M,B
  END SUBROUTINE REAL_LINE_EQ
  SUBROUTINE INT_LINE_EQ(X1,Y1,X2,Y2,M,B)
    INTEGER,INTENT(IN) :: X1,Y1,X2,Y2
    INTEGER,INTENT(OUT) :: M,B
  END SUBROUTINE INT_LINE_EQ
END INTERFACE
```

In this example, LINE_EQUATION is the generic name which can be used for either REAL_LINE_EQ or INT_LINE_EQ. Fortran selects the appropriate subroutine according to the nature of the arguments passed to LINE_EQUATION. Even when a generic name exists, you can always invoke a procedure by its specific name. In the previous example, you can call REAL_LINE_EQ by its specific name (REAL_LINE_EQ), or its generic name LINE_EQUATION.

See Also

- Procedure Interfaces
- INTERFACE

Defining Generic Operators

An interface block can be used to define a generic operator. The only procedures allowed in the interface block are functions that can be referenced as defined operations.

The initial line for such an interface block takes the following form:

INTERFACE OPERATOR (*op*)

op Is one of the following:

- A defined unary operator (one argument)
- A defined binary operator (two arguments)
- An extended intrinsic operator (number of arguments must be consistent with the intrinsic uses of that operator)

The functions within the interface block must have one or two nonoptional arguments with intent IN, and the function result must not be of type character with assumed length. A defined operation is treated as a reference to the function.

The following shows the form (and an example) of a defined unary and defined binary operation:

Operation	Form	Example
Defined Unary	.defined-operator. operand[1]	.MINUS. C
Defined Binary	operand[2] .defined-operator. operand[3]	B .MINUS. C

[1] The operand corresponds to the function's dummy argument.

[2] The left operand corresponds to the first dummy argument of the function.

[3] The right operand corresponds to the second argument.

For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Both forms of each relational operator have the same interpretation, so extending one form (such as >=) defines both forms (>= and .GE.).

The following is an example of a procedure interface block defining a new operator:

```
INTERFACE OPERATOR(.BAR.)
  FUNCTION BAR(A_1)
    INTEGER, INTENT(IN) :: A_1
    INTEGER :: BAR
  END FUNCTION BAR
END INTERFACE
```

The following example shows a way to reference function BAR by using the new operator:

```
INTEGER B
I = 4 + (.BAR. B)
```

The following is an example of a procedure interface block with a defined operator extending an existing operator:

```
INTERFACE OPERATOR(+)
  FUNCTION LGFUNC (A, B)
    LOGICAL, INTENT(IN) :: A(:), B(SIZE(A))
    LOGICAL :: LGFUNC(SIZE(A))
  END FUNCTION LGFUNC
END INTERFACE
```

The following example shows two equivalent ways to reference function LGFUNC:

```
LOGICAL, DIMENSION(1:10) :: C, D, E
N = 10
E = LGFUNC(C(1:N), D(1:N))
E = C(1:N) + D(1:N)
```

See Also

- Procedure Interfaces
- INTENT attribute
- INTERFACE
- Expressions
- Defined Operations

Defining Generic Assignment

An interface block can be used to define generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.

The initial line for such an interface block takes the following form:

INTERFACE ASSIGNMENT (=)

The subroutines within the interface block must have two nonoptional arguments, the first with intent OUT or INOUT, and the second with intent IN.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC
  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                   ! Contains definition of type STRING
    TYPE(STRING), INTENT(OUT) :: STR    ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
CALL CHAR_TO_STRING(CH, '432C')
CH = '432C'
```

See Also

- Procedure Interfaces
- Defined Assignments
- INTENT attribute
- INTERFACE statement

CONTAINS Statement Overview

This statement introduces internal or module procedures. For more information, see CONTAINS.

ENTRY Statement Overview

The ENTRY statement provides multiple entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram. For more information, see ENTRY.

See Also

- Program Units and Procedures
- ENTRY Statements in Function Subprograms
- ENTRY Statements in Subroutine Subprograms
- ENTRY Statements in Function Subprograms
- ENTRY Statements in Subroutine Subprograms

ENTRY Statements in Function Subprograms

If the ENTRY statement is contained in a function subprogram, it defines an additional function. The name of the function is the name specified in the ENTRY statement, and its result variable is the entry name or the name specified by RESULT (if any).

If the entry result variable has the same characteristics as the FUNCTION statement's result variable, their result variables identify the same variable, even if they have different names. Otherwise, the result variables are storage associated and must all be nonpointer scalars of intrinsic type, in one of the following groups:

Group 1	Type default integer, default real, double precision real, default complex, double complex, or default logical
Group 2	Type REAL(16) and COMPLEX(16)
Group 3	Type default character (with identical lengths)

All entry names within a function subprogram are associated with the name of the function subprogram. Therefore, defining any entry name or the name of the function subprogram defines all the associated names with the same data type. All associated names with different data types become undefined.

If RESULT is specified in the ENTRY statement and RECURSIVE is specified in the FUNCTION statement, the interface of the function defined by the ENTRY statement is explicit within the function subprogram.

Examples

The following example shows a function subprogram that computes the hyperbolic functions SINH, COSH, and TANH:

```
REAL FUNCTION TANH(X)

  TSINH(Y) = EXP(Y) - EXP(-Y)

  TCOSH(Y) = EXP(Y) + EXP(-Y)

  TANH = TSINH(X)/TCOSH(X)

  RETURN

  ENTRY SINH(X)

  SINH = TSINH(X)/2.0

  RETURN

  ENTRY COSH(X)

  COSH = TCOSH(X)/2.0

  RETURN

END
```

See Also

- ENTRY Statement Overview
- RESULT keyword

ENTRY Statements in Subroutine Subprograms

If the ENTRY statement is contained in a subroutine subprogram, it defines an additional subroutine. The name of the subroutine is the name specified in the ENTRY statement.

If RECURSIVE is specified on the SUBROUTINE statement, the interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

Examples

The following example shows a main program calling a subroutine containing an ENTRY statement:

```
PROGRAM TEST

  ...

  CALL SUBA(A, B, C)       ! A, B, and C are actual arguments

  ...                      !    passed to entry point SUBA

END

SUBROUTINE SUB(X, Y, Z)

  ...

  ENTRY SUBA(Q, R, S)      ! Q, R, and S are dummy arguments

  ...                      ! Execution starts with this statement

END SUBROUTINE
```

The following example shows an ENTRY statement specifying alternate returns:

```
CALL SUBC(M, N, *100, *200, P)

...

SUBROUTINE SUB(K, *, *)

  ...

  ENTRY SUBC(J, K, *, *, X)

  ...

  RETURN 1

  RETURN 2

END
```

Note that the CALL statement for entry point SUBC includes actual alternate return arguments. The RETURN 1 statement transfers control to statement label 100 and the RETURN 2 statement transfers control to statement label 200 in the calling program.

IMPORT Statement Overview

The IMPORT statement makes host entities accessible in the interface body of an interface block. For more information, see IMPORT.

Intrinsic Procedures

50

Intrinsic procedures are functions and subroutines that are included in the Fortran library. There are four classes of these intrinsic procedures, as follows:

- Elemental procedures

  These procedures have scalar dummy arguments that can be called with scalar or array actual arguments. There are many elemental intrinsic functions and one elemental intrinsic subroutine (MVBITS).

  If the arguments are all scalar, the result is scalar. If an actual argument is array-valued, the intrinsic procedure is applied to each element of the actual argument, resulting in an array that has the same shape as the actual argument.

  If there is more than one array-valued argument, they must all have the same shape.

  Many algorithms involving arrays can now be written conveniently as a series of computations with whole arrays. For example, consider the following:

  ```
  a = b + c

  ...               ! a, b, c, and s are all arrays of similar shape

  s = sum(a)
  ```

  The above statements can replace entire DO loops.

  Consider the following:

  ```
  real, dimension (5,5) x,y

  . . .      !Assign values to x.

  y = sin(x) !Pass the entire array as an argument.
  ```

  In this example, since the SIN(X) function is an elemental procedure, it operates element-by-element on the array x when you pass it the name of the whole array.

- Inquiry functions

  These functions have results that depend on the properties of their principal argument, not the value of the argument (the argument value can be undefined).

- Transformational functions

  These functions have one or more array-valued dummy or actual arguments, an array result, or both. The intrinsic function is not applied elementally to an array-valued actual argument; instead it changes (transforms) the argument array into another array.

- Nonelemental procedures

These procedures must be called with only scalar arguments; they return scalar results. All subroutines (except MVBITS) are nonelemental.

Intrinsic procedures are invoked the same way as other procedures, and follow the same rules of argument association.

The intrinsic procedures have generic (or common) names, and many of the intrinsic functions have specific names. (Some intrinsic functions are both generic and specific.)

In general, generic functions accept arguments of more than one data type; the data type of the result is the same as that of the arguments in the function reference. For elemental functions with more than one argument, all arguments must be of the same type (except for the function MERGE).

When an intrinsic function is passed as an actual argument to a procedure, its specific name must be used, and when called, its arguments must be scalar. Some specific intrinsic functions are not allowed as actual arguments in all circumstances. The following table lists specific functions that cannot be passed as actual arguments.

Table 595: Specific Intrinsic Functions Not Allowed as Actual Arguments

AIMAX0	FLOATI	IZEXT	LOC
AIMIN0	FLOATJ	JFIX	MAX0
AJMAX0	FLOATK	JIDINT	MAX1
AJMIN0	FP_CLASS	JIFIX	MIN0
AKMAX0	HFIX	JINT	MIN1
AKMIN0	IADDR	JIQINT	MULT_HIGH
AMAX0	IARGC	JMAX0	MULT_HIGH_SIGNED
AMAX1	ICHAR	JMAX1	NARGS
AMIN0	IDINT	JMIN0	QCMPLX
AMIN1	IFIX	JMIN1	QEXT
CHAR	IIDINT	JNUM	QEXTD
CMPLX	IIFIX	JZEXT	QMAX1
DBLE	IINT	KIDINT	QMIN1
DBLEQ	IIQINT	KIFIX	QNUM

DCMPLX	IJINT	KINT	QREAL
DFLOTI	IMAX0	KIQINT	RAN
DFLOTJ	IMAX1	KMAX0	REAL
DFLOTK	IMIN0	KMAX1	RNUM
DMAX1	IMIN1	KMIN0	SECNDS
DMIN1	INT	KMIN1	SHIFTL
DNUM	INT1	KNUM	SHIFTR
DPROD	INT2	KZEXT	SNGL
DREAL	INT4	LGE	SNGLQ
DSHIFTL	INT8	LGT	ZEXT
DSHIFTR	INUM	LLE	
FLOAT	IQINT	LLT	

Note that none of the intrinsic subroutines can be passed as actual arguments.

This chapter also contains information on the following topics:

- Argument keywords in intrinsic procedures
- Overview of bit functions
- Categories and Lists of intrinsic procedures

The A to Z Reference contains the descriptions of all intrinsics listed in alphabetical order. Each reference entry indicates whether the procedure is inquiry, elemental, transformational, or nonelemental, and whether it is a function or a subroutine.

Argument Keywords in Intrinsic Procedures

For all intrinsic procedures, the arguments shown are the names you must use as keywords when using the keyword form for actual arguments. For example, a reference to function CMPLX(X, Y, KIND) can be written as follows:

Using positional arguments:	CMPLX(F, G, L)

Using argument keywords: [1]	CMPLX(KIND=L, Y=G, X=F)

[1] Note that argument keywords can be written in any order.

Some argument keywords are optional (denoted by square brackets). The following describes some of the most commonly used optional arguments:

BACK	Specifies that a string scan is to be in reverse order (right to left).
DIM	Specifies a selected dimension of an array argument.
KIND	Specifies the kind type parameter of the function result.
MASK	Specifies that a mask can be applied to the elements of the argument array to exclude the elements that are not to be involved in an operation.

Examples

The syntax for the DATE_AND_TIME intrinsic subroutine shows four optional positional arguments: DATE, TIME, ZONE, and VALUES. The following shows some valid ways to specify these arguments:

```
! Keyword example

CALL DATE_AND_TIME (ZONE=Z)

! Positional example
CALL DATE_AND_TIME (DATE, TIME, ZONE)
```

See Also

- Intrinsic Procedures
- CALL
- Function references
- Argument Association

Overview of Bit Functions

Integer data types are represented internally in binary two's complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0.

The intrinsic functions IAND, IOR, IEOR, and NOT operate on all of the bits of their argument (or arguments). Bit 0 of the result comes from applying the specified logical operation to bit 0 of the argument. Bit 1 of the result comes from applying the specified logical operation to bit 1 of the argument, and so on for all of the bits of the result.

The functions ISHFT and ISHFTC shift binary patterns.

The functions IBSET, IBCLR, BTEST, and IBITS and the subroutine MVBITS operate on bit fields.

A *bit field* is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 47 is represented by the following:

Binary pattern:	0...0101111
Bit position:	n...6543210 Where *n* is the number of bit positions in the numeric storage unit.

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4.

Negative integers are represented in two's complement notation. For example, the integer -47 is represented by the following:

Binary pattern:	1...1010001
Bit position:	n...6543210 Where n is the number of bit positions in the numeric storage unit.

The value of bit position n is as follows:

```
1 for a negative number
0 for a non-negative number
```

All the high-order bits in the pattern from the last significant bit of the value up to bit n are the same as bit n.

IBITS and MVBITS operate on general bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate on 1-bit fields. They do not require a length argument.

For IBSET, IBCLR, and BTEST, the bit position range is as follows:

- 0 to 63 for INTEGER(8) and LOGICAL(8)
- 0 to 31 for INTEGER(4) and LOGICAL(4)
- 0 to 15 for INTEGER(2) and LOGICAL(2)
- 0 to 7 for BYTE, INTEGER(1), and LOGICAL(1)

For IBITS, the bit position can be any number. The length range is 0 to 63 on Intel® 64 architecture and IA-64 architecture; 0 to 31 on IA-32 architecture.

The following example shows IBSET, IBCLR, and BTEST:

```
I = 4
J = IBSET (I,5)
PRINT *, 'J = ',J
K = IBCLR (J,2)
PRINT *, 'K = ',K
PRINT *, 'Bit 2 of K is ',BTEST(K,2)
END
```

The results are: J = 36, K = 32, and Bit 2 of K is F.

For optimum selection of performance and memory requirements, Intel Fortran provides the following integer data types:

Data Type	Storage Required (in bytes)
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8

The bit manipulation functions each have a generic form that operates on all of these integer types and a specific form for each type.

When you specify the intrinsic functions that refer to bit positions or that shift binary patterns within a storage unit, be careful that you do not create a value that is outside the range of integers representable by the data type. If you shift by an amount greater than or equal to the size of the object you're shifting, the result is 0.

Consider the following:

```
INTEGER(2) I,J

I = 1

J = 17

I = ISHFT(I,J)
```

The variables I and J have INTEGER(2) type. Therefore, the generic function ISHFT maps to the specific function IISHFT, which returns an INTEGER(2) result. INTEGER(2) results must be in the range -32768 to 32767, but the value 1, shifted left 17 positions, yields the binary pattern 1 followed by 17 zeros, which represents the integer 131072. In this case, the result in I is 0.

The previous example would be valid if I was INTEGER(4), because ISHFT would then map to the specific function JISHFT, which returns an INTEGER(4) value.

If ISHFT is called with a constant first argument, the result will either be the default integer size or the smallest integer size that can contain the first argument, whichever is larger.

Categories and Lists of Intrinsic Procedures

This section describes the categories of generic intrinsic functions (including a summarizing table) and lists the intrinsic subroutines.

Intrinsic procedures are fully described (in alphabetical order) in the A to Z Reference.

Categories of Intrinsic Functions

Generic intrinsic functions can be divided into categories, as shown in the following table:

Table 601: Categories of Intrinsic Functions

Category	Subcategory	Description
Numeric	Computation	Elemental functions that perform type conversions or simple numeric operations: ABS, AIMAG, AINT, AMAX0, AMIN0, ANINT, CEILING, CMPLX, CONJG, DBLE, DCMPLX, DFLOAT, DIM, DNUM, DPROD, DREAL,

Category	Subcategory	Description
		FLOAT, FLOOR, IFIX, IMAG, INT, INUM, JNUM, KNUM MAX, MAX1, MIN, MIN1, MOD, MODULO, NINT, QCMPLX, QEXT, QFLOAT, QNUM, QREAL, REAL, RNUM, SIGN, SNGL, ZEXT
		Nonelemental function that provides a pseudorandom number RAN
	Manipulation[1]	Elemental functions that return values related to the components of the model values associated with the actual value of the argument: EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING
	Inquiry[1]	Functions that return scalar values from the models associated with the type and kind parameters of their arguments[2]: DIGITS, EPSILON, HUGE, ILEN, MAXEXPONENT, MINEXPONENT, PRECISION, RADIX, RANGE, SIZEOF, TINY
	Transformational	Functions that perform vector and matrix multiplication: DOT_PRODUCT, MATMUL
	System	Functions that return information about a process or processor: MCLOCK, SECNDS

1954

Category	Subcategory	Description
Kind type		Functions that return kind type parameters: KIND, SELECTED_CHAR_KIND, SELECTED_INT_KIND, SELECTED_REAL_KIND
Mathematical		Elemental functions that perform mathematical operations: ACOS, ACOSD, ACOSH, ASIN, ASIND, ASINH, ATAN, ATAN2, ATAN2D, ATAND, ATANH, COS, COSD, COSH, COTAN, COTAND, EXP, LOG, LOG10, SIN, SIND, SINH, SQRT, TAN, TAND, TANH
Bit	Manipulation	Elemental functions that perform single-bit processing, logical and shift operations, and allow bit subfields to be referenced: AND, BTEST, DSHIFTL, DSHIFTR, IAND, IBCHNG, IBCLR, IBITS, IBSET, IEOR, IOR, ISHA, ISHC, ISHFT, ISHFTC, ISHL, IXOR, LSHIFT, NOT, OR, RSHIFT, SHIFTL, SHIFTR, XOR
	Inquiry	Function that lets you determine parameter s (the bit size) in the bit model[3]: BIT_SIZE
	Representation	Elemental functions that return information on bit representation of integers: LEADZ, POPCNT, POPPAR, TRAILZ

Category	Subcategory	Description
Character	Comparison	Elemental functions that make a lexical comparison of the character-string arguments and return a default logical result: LGE, LGT, LLE, LLT
	Conversion	Elemental functions that take character arguments and return integer, ASCII, or character values[4]: ACHAR, CHAR, IACHAR, ICHAR
	String handling	Functions that perform operations on character strings, return lengths of arguments, and search for certain arguments: Elemental: ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, SCAN, VERIFY; Nonelemental: REPEAT, TRIM
	Inquiry	Functions that return the length of an argument or information about command-line arguments: COMMAND_ARGUMENT_COUNT, IARG, IARGC, LEN, NARGS, NUMARG
Array	Construction	Functions that construct new arrays from the elements of existing arrays: Elemental: MERGE; Nonelemental: PACK, SPREAD, UNPACK
	Inquiry	Functions that let you determine if an array argument is allocated, and return the size or shape of an array, and the lower and

1956

Category	Subcategory	Description
		upper bounds of subscripts along each dimension: ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND
	Location	Transformational functions that find the geometric locations of the maximum and minimum values of an array: MAXLOC, MINLOC
	Manipulation	Transformational functions that shift an array, transpose an array, or change the shape of an array: CSHIFT, EOSHIFT, RESHAPE, TRANSPOSE
	Reduction	Transformational functions that perform operations on arrays. The functions "reduce" elements of a whole array to produce a scalar result, or they can be applied to a specific dimension of an array to produce a result array with a rank reduced by one: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM
Miscellaneous		Functions that do the following: • Check for pointer association (ASSOCIATED) • Return an address (BADDRESS or IADDR) • Return the size of a level of the memory cache (CACHESIZE)[5]

Category	Subcategory	Description
		• Check for end-of-file (EOF) • Return error functions (ERF and ERFC) • Return the class of a floating-point argument (FP_CLASS) • Return the INTEGER KIND that will hold an address (INT_PTR_KIND) • Test for Not-a-Number values (ISNAN) • Return the internal address of a storage item (LOC) • Return a logical value of an argument (LOGICAL) • Allocate memory (MALLOC) • Return the upper 64 bits of a 128-bit unsigned result (MULT_HIGH)[5] • Return the upper 64 bits of a 128-bit signed result (MULT_HIGH_SIGNED)[5] • Return a new line character (NEW_LINE) • Return a disassociated pointer (NULL) • Check for argument presence (PRESENT) • Convert a bit pattern (TRANSFER) • Check for end-of-file condition (IS_IOSTAT_END)

Category	Subcategory	Description
		• Check for end-of-record condition (IS_IOSTAT_EOR)

[1] All of the numeric manipulation, and many of the numeric inquiry functions are defined by the model sets for integers and reals.

[2] The value of the argument does not have to be defined.

[3] For more information on bit functions, see Bit functions.

[4] The Intel® Fortran processor character set is ASCII, so ACHAR = CHAR and IACHAR = ICHAR.

[5] i64 only

The following table summarizes the generic intrinsic functions and indicates whether they are elemental, inquiry, or transformational functions. Optional arguments are shown within square brackets.

Some intrinsic functions are specific with no generic association. These functions are listed below.

Table 602: Summary of Generic Intrinsic Functions

Generic Function	Class	Value Returned
ABS (A)	E	The absolute value of an argument
ACHAR (I)	E	The character in the specified position of the ASCII character set
ACOS (X)	E	The arccosine (in radians) of the argument
ACOSD (X)	E	The arccosine (in degrees) of the argument
ACOSH (X)	E	The hyperbolic arccosine of the argument

Generic Function	Class	Value Returned
ADJUSTL (STRING)	E	The specified string with leading blanks removed and placed at the end of the string
ADJUSTR (STRING)	E	The specified string with trailing blanks removed and placed at the beginning of the string
AIMAG (Z)	E	The imaginary part of a complex argument
AINT (A [,KIND])	E	A real value truncated to a whole number
ALL (MASK [,DIM])	T	.TRUE. if all elements of the masked array are true
ALLOCATED (ARRAY)	I	The allocation status of the argument array
AMAX0 (A1, A2 [, A3,...])	E	The maximum value in a list of integers (returned as a real value)
AMIN0 (A1, A2 [, A3,...])	E	The minimum value in a list of integers (returned as a real value)
AND (I, J)	E	See IAND
ANINT (A [, KIND])	E	A real value rounded to a whole number
ANY (MASK [, DIM])	T	.TRUE. if any elements of the masked array are true
ASIN (X)	E	The arcsine (in radians) of the argument

Generic Function	Class	Value Returned
ASIND (X)	E	The arcsine (in degrees) of the argument
ASINH (X)	E	The hyperbolic arcsine of the argument
ASSOCIATED (POINTER [,TARGET])	I	.TRUE. if the pointer argument is associated or the pointer is associated with the specified target
ATAN (X)	E	The arctangent (in radians) of the argument
ATAN2 (Y, X)	E	The arctangent (in radians) of the arguments
ATAN2D (Y, X)	E	The arctangent (in degrees) of the arguments
ATAND (X)	E	The arctangent (in degrees) of the argument
ATANH (X)	E	The hyperbolic arctangent of the argument
BADDRESS (X)	I	The address of the argument
BIT_SIZE (I)	I	The number of bits (s) in the bit model
BTEST (I, POS)	E	.TRUE. if the specified position of argument I is one
CEILING (A [,KIND])	E	The smallest integer greater than or equal to the argument value
CHAR (I [,KIND])	E	The character in the specified position of the processor character set

Generic Function	Class	Value Returned
COMMAND_ARGUMENT_COUNT ()	I	The number of command arguments
CONJG (Z)	E	The conjugate of a complex number
COS (X)	E	The cosine of the argument, which is in radians
COSD (X)	E	The cosine of the argument, which is in degrees
COSH (X)	E	The hyperbolic cosine of the argument
COTAN (X)	E	The cotangent of the argument, which is in radians
COTAND (X)	E	The cotangent of the argument, which is in degrees
COUNT (MASK [,DIM] [,KIND])	T	The number of .TRUE. elements in the argument array
CSHIFT (ARRAY, SHIFT [,DIM])	T	An array that has the elements of the argument array circularly shifted
DBLE (A)	E	The corresponding double precision value of the argument
DFLOAT (A)	E	The corresponding double precision value of the integer argument
DIGITS (X)	I	The number of significant digits in the model for the argument

Generic Function	Class	Value Returned
DIM (X, Y)	E	The positive difference between the two arguments
DOT_PRODUCT (VECTOR_A, VECTOR_B)	T	The dot product of two rank-one arrays (also called a vector multiply function)
DREAL (A)	E	The corresponding double-precision value of the double complex argument
DSHIFTL (ILEFT, IRIGHT, ISHIFT)	E	The upper (leftmost) 64 bits of a left-shifted 128-bit integer
DSHIFTR (ILEFT, IRIGHT, ISHIFT)	E	The lower (rightmost) 64 bits of a right-shifted 128-bit integer
EOF (A)	I	.TRUE. or .FALSE. depending on whether a file is beyond the end-of-file record
EOSHIFT (ARRAY, SHIFT [,BOUNDARY] [,DIM])	T	An array that has the elements of the argument array end-off shifted
EPSILON (X)	I	The number that is almost negligible when compared to one
ERF (X)	E	The error function of an argument
ERFC (X)	E	The complementary error function of an argument
EXP (X)	E	The exponential e^x for the argument x
EXPONENT (X)	E	The value of the exponent part of a real argument

Generic Function	Class	Value Returned
FLOAT (X)	E	The corresponding real value of the integer argument
FLOOR (A [,KIND])	E	The largest integer less than or equal to the argument value
FP_CLASS (X)	E	The class of the IEEE floating-point argument
FRACTION (X)	E	The fractional part of a real argument
HUGE (X)	I	The largest number in the model for the argument
IACHAR (C)	E	The position of the specified character in the ASCII character set
IADDR (X)	E	See BADDRESS
IAND (I, J)	E	The logical AND of the two arguments
IBCLR (I, POS)	E	The specified position of argument I cleared (set to zero)
IBCHNG (I, POS)	E	The reversed value of a specified bit
IBITS (I, POS, LEN)	E	The specified substring of bits of argument I
IBSET (I, POS)	E	The specified bit in argument I set to one
ICHAR (C [, KIND])	E	The position of the specified character in the processor character set

1964

Generic Function	Class	Value Returned
IEOR (I, J)	E	The logical exclusive OR of the corresponding bit arguments
IFIX (X)	E	The corresponding integer value of the real argument rounded as if it were an implied conversion in an assignment
ILEN (I)	I	The length (in bits) in the two's complement representation of an integer
IMAG (Z)	E	See AIMAG
INDEX (STRING, SUBSTRING [, BACK] [,KIND])	E	The position of the specified substring in a character expression
INT (A [, KIND])	E	The corresponding integer value (truncated) of the argument
IOR (I, J)	E	The logical inclusive OR of the corresponding bit arguments
ISHA (I, SHIFT)	E	Argument I shifted left or right by a specified number of bits
ISHC (I, SHIFT)	E	Argument I rotated left or right by a specified number of bits
ISHFT (I, SHIFT)	E	The logical end-off shift of the bits in argument I
ISHFTC (I, SHIFT [,SIZE])	E	The logical circular shift of the bits in argument I

Generic Function	Class	Value Returned
ISHL (I, SHIFT)	E	Argument I logically shifted left or right by a specified number of bits
ISNAN (X)	E	Tests for Not-a-Number (NaN) values
IXOR (I, J)	E	See IEOR
KIND (X)	I	The kind type parameter of the argument
LBOUND (ARRAY [,DIM] [,KIND])	I	The lower bounds of an array (or one of its dimensions)
LEADZ (I)	E	The number of leading zero bits in an integer
LEN (STRING [,KIND])	I	The length (number of characters) of the argument character string
LEN_TRIM (STRING [,KIND])	E	The length of the specified string without trailing blanks
LGE (STRING_A, STRING_B)	E	A logical value determined by a > or = comparison of the arguments
LGT (STRING_A, STRING_ B)	E	A logical value determined by a > comparison of the arguments
LLE (STRING_A, STRING_B)	E	A logical value determined by a < or = comparison of the arguments
LLT (STRING_A, STRING_B)	E	A logical value determined by a < comparison of the arguments

1966

Generic Function	Class	Value Returned
LOC (A)	I	The internal address of the argument.
LOG (X)	E	The natural logarithm of the argument
LOG10 (X)	E	The common logarithm (base 10) of the argument
LOGICAL (L [,KIND])	E	The logical value of the argument converted to a logical of type KIND
LSHIFT (I, POSITIVE_SHIFT)	E	See ISHFT
LSHFT (I, POSITIVE_SHIFT)	E	Same as LSHIFT; see ISHFT
MALLOC (I)	E	The starting address for the block of memory allocated
MATMUL (MATRIX_A, MATRIX_B)	T	The result of matrix multiplication (also called a matrix multiply function)
MAX (A1, A2 [, A3,...])	E	The maximum value in the set of arguments
MAX1 (A1, A2 [, A3,...])	E	The maximum value in the set of real arguments (returned as an integer)
MAXEXPONENT (X)	I	The maximum exponent in the model for the argument
MAXLOC (ARRAY [,DIM] [,MASK] [,KIND])	T	The rank-one array that has the location of the maximum element in the argument array
MAXVAL (ARRAY [,DIM] [,MASK])	T	The maximum value of the elements in the argument array

Generic Function	Class	Value Returned
MERGE (TSOURCE, FSOURCE, MASK)	E	An array that is the combination of two conformable arrays (under a mask)
MIN (A1, A2 [, A3,...])	E	The minimum value in the set of arguments
MIN1 (A1, A2 [, A3,...])	E	The minimum value in the set of real arguments (returned as an integer)
MINEXPONENT (X)	I	The minimum exponent in the model for the argument
MINLOC (ARRAY [,DIM] [,MASK] [,KIND])	T	The rank-one array that has the location of the minimum element in the argument array
MINVAL (ARRAY [,DIM] [,MASK])	T	The minimum value of the elements in the argument array
MOD (A, P)	E	The remainder of the arguments (has the sign of the first argument)
MODULO (A, P)	E	The modulo of the arguments (has the sign of the second argument)
NEAREST (X, S)	E	The nearest different machine-representable number in a given direction
NEW_LINE (A)	I	A new line character
NINT (A [,KIND])	E	A real value rounded to the nearest integer

Generic Function	Class	Value Returned
NOT (I)	E	The logical complement of the argument
NULL ([MOLD])	T	A disassociated pointer
OR (I, J)	E	See IOR
PACK (ARRAY, MASK [,VECTOR])	T	A packed array of rank one (under a mask)
POPCNT (I)	E	The number of 1 bits in the integer argument
POPPAR (I)	E	The parity of the integer argument
PRECISION (X)	I	The decimal precision (real or complex) of the argument
PRESENT (A)	I	.TRUE. if an actual argument has been provided for an optional dummy argument
PRODUCT (ARRAY [,DIM] [,MASK])	T	The product of the elements of the argument array
QEXT (A)	E	The corresponding REAL(16) precision value of the argument
QFLOAT (A)	E	The corresponding REAL(16) precision value of the integer argument
RADIX (X)	I	The base of the model for the argument
RANGE (X)	I	The decimal exponent range of the model for the argument

Generic Function	Class	Value Returned
REAL (A [, KIND])	E	The corresponding real value of the argument
REPEAT (STRING, NCOPIES)	T	The concatenation of zero or more copies of the specified string
RESHAPE (SOURCE, SHAPE [,PAD] [,ORDER])	T	An array that has a different shape than the argument array, but the same elements
RRSPACING (X)	E	The reciprocal of the relative spacing near the argument
RSHIFT (I, NEGATIVE_SHIFT)	E	See ISHFT
RSHFT (I, NEGATIVE_SHIFT)	E	Same as RSHIFT; see ISHFT
SCALE (X, I)	E	The value of the exponent part (of the model for the argument) changed by a specified value
SCAN (STRING, SET [,BACK] [,KIND])	E	The position of the specified character (or set of characters) within a string
SELECTED_CHAR_KIND (NAME)	T	The value of the kind type parameter of the character set named by the argument
SELECTED_INT_KIND (R)	T	The integer kind parameter of the argument
SELECTED_REAL_KIND ([P] [, R])	T	The real kind parameter of the argument; one of the optional arguments must be specified

1970

Generic Function	Class	Value Returned
SET_EXPONENT (X, I)	E	The value of the exponent part (of the model for the argument) set to a specified value
SHAPE (SOURCE [,KIND])	I	The shape (rank and extents) of an array or scalar
SHIFTL (IVALUE, ISHIFT)	E	Argument IVALUE shifted left by a specified number of bits
SHIFTR (IVALUE, ISHIFT)	E	Argument IVALUE shifted right by a specified number of bits
SIGN (A, B)	E	A value with the sign transferred from its second argument
SIN (X)	E	The sine of the argument, which is in radians
SIND (X)	E	The sine of the argument, which is in degreess
SINH (X)	E	The hyperbolic sine of the argument
SIZE (ARRAY [,DIM] [,KIND])	I	The size (total number of elements) of the argument array (or one of its dimensions)
SIZEOF (X)	I	The bytes of storage used by the argument
SNGL (X)	E	The corresponding real value of the argument
SPACING (X)	E	The value of the absolute spacing of model numbers near the argument

Generic Function	Class	Value Returned
SPREAD (SOURCE, DIM, NCOPIES)	T	A replicated array that has an added dimension
SQRT (X)	E	The square root of the argument
SUM (ARRAY [,DIM] [,MASK])	T	The sum of the elements of the argument array
TAN (X)	E	The tangent of the argument, which is in radians
TAND (X)	E	The tangent of the argument, which is in degrees
TANH (X)	E	The hyperbolic tangent of the argument
TINY (X)	I	The smallest positive number in the model for the argument
TRAILZ (I)	E	The number of trailing zero bits in an integer
TRANSFER (SOURCE, MOLD [,SIZE])	T	The bit pattern of SOURCE converted to the type and kind parameters of MOLD
TRANSPOSE (MATRIX)	T	The matrix transpose for the rank-two argument array
TRIM (STRING)	T	The argument with trailing blanks removed
UBOUND (ARRAY [,DIM] [,KIND])	I	The upper bounds of an array (or one of its dimensions)
UNPACK (VECTOR, MASK, FIELD)	T	An array (under a mask) unpacked from a rank-one array

1972

Generic Function	Class	Value Returned
VERIFY (STRING, SET [,BACK] [,KIND])	E	The position of the first character in a string that does not appear in the given set of characters
XOR (I, J)	E	See IEOR
ZEXT (X [,KIND])	E	A zero-extended value of the argument
Key to Classes		
E-Elemental		
I-Inquiry		
T-Transformational		

The following table lists specific functions that have no generic function associated with them and indicates whether they are elemental, nonelemental, or inquiry functions. Optional arguments are shown within square brackets.

Table 603: Specific Functions with No Generic Association

Generic Function	Class	Value Returned
CACHESIZE (N)[1]	I	The size of a level of the memory cache
CMPLX (X [,Y] [,KIND])	E	The corresponding complex value of the argument
DCMPLX (X, Y)	E	The corresponding double complex value of the argument
DNUM (I)	E	The corresponding REAL(8) value of a character string
DPROD (X, Y)	E	The double-precision product of two real arguments
DREAL (A)	E	The corresponding double-precision value of the double-complex argument

Generic Function	Class	Value Returned
IARG ( )	I	See IARGC
IARGC ( )	I	The index of the last command-line argument
INT_PTR_KIND ( )	I	The INTEGER kind that will hold an address
INUM (I)	E	The corresponding INTEGER(2) value of a character string
JNUM (I)	E	The corresponding INTEGER(4) value of a character string
KNUM (I)	E	The corresponding INTEGER(8) value of a character string
MCLOCK ( )	I	The sum of the current process's user time and the user and system time of all its child processes
MULT_HIGH (I, J)[1]	E	The upper (leftmost) 64 bits of the 128-bit unsigned result
MULT_HIGH_SIGNED (I, J)[1]	E	The upper (leftmost) 64 bits of the 128-bit signed result
NARGS ( )	I	The total number of command-line arguments, including the command
NUMARG ( )	I	See IARGC
QCMPLX (X, Y)	E	The corresponding COMPLEX(16) value of the argument

1974

Generic Function	Class	Value Returned
QNUM (I)	E	The corresponding REAL(16) value of a character string
QREAL (A)	E	The corresponding REAL(16) value of the real part of a COMPLEX(16) argument
RAN (I)	N	The next number from a sequence of pseudorandom numbers (uniformly distributed in the range 0 to 1)
RNUM (I)	E	The corresponding REAL(4) value of a character string
SECNDS (X)	E	The system time of day (or elapsed time) as a floating-point value in seconds

Key to Classes

E-Elemental

I-Inquiry

N-Nonelemental

[1]i64 only

Intrinsic Subroutines

The following table lists the intrinsic subroutines. Optional arguments are shown within square brackets. All these subroutines are nonelemental except for MVBITS. None of the intrinsic subroutines can be passed as actual arguments.

Table 604: Intrinsic Subroutines

Subroutine	Value Returned or Result
CPU_TIME (TIME)	The processor time in seconds

Subroutine	Value Returned or Result
DATE (BUF)	The ASCII representation of the current date (in dd-mmm-yy form)
DATE_AND_TIME ([DATE] [,TIME] [,ZONE] [,VALUES])	Date and time information from the real-time clock
ERRSNS ([IO_ERR] [,SYS_ERR] [,STAT] [,UNIT] [,COND])	Information about the most recently detected error condition
EXIT ([STATUS])	Image exit status is optionally returned; the program is terminated, all files closed, and control is returned to the operating system
FREE (A)	Frees memory that is currently allocated
GETARG (N, BUFFER [,STATUS])	The specified command line argument (where the command itself is argument number zero)
GET_COMMAND ([command, length, status])	The entire command that was used to invoke the program
GET_COMMAND_ARGUMENT (n [, value, length, status])	A command line argument of the command that invoked the program
GET_ENVIRONMENT_VARIABLE (name [, value, length, status, trim_name])	The value of an environment variable
IDATE (I, J, K)	Three integer values representing the current month, day, and year
MM_PREFETCH (ADDRESS [,HINT] [,FAULT] [,EXCLUSIVE])	Data from the specified address on one memory cache line
MOVE_ALLOC (FROM, TO)	An allocation is moved from one allocatable object to another.
MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)[1]	A sequence of bits (bit field) is copied from one location to another

1976

Subroutine	Value Returned or Result
RANDOM_NUMBER (HARVEST)	A pseudorandom number taken from a sequence of pseudorandom numbers uniformly distributed within the range 0.0 to 1.0
RANDOM_SEED ([SIZE] [,PUT] [,GET])	The initialization or retrieval of the pseudorandom number generator seed value
RANDU (I1, I2, X)	A pseudorandom number as a single-precision value (within the range 0.0 to 1.0)
SYSTEM_CLOCK ([COUNT] [,COUNT_RATE] [,COUNT_MAX])	Data from the processors real-time clock
TIME (BUF)	The ASCII representation of the current time (in hh:mm:ss form)

[1] An elemental subroutine

Data Transfer I/O Statements 51

Input/Output (I/O) statements can be used for data transfer, file connection, file inquiry, and file positioning.

This section discusses data transfer and contains information on the following topics:

- An overview of records and files
- Components of data transfer statements
- Data transfer input statements:
  - READ statement
  - ACCEPT statement
- Data transfer output statements:
  - WRITE statement
  - PRINT and TYPE statements
  - REWRITE statement

File connection, file inquiry, and file positioning I/O statements are discussed in File Operation I/O Statements.

Records and Files

A record is a sequence of values or a sequence of characters. There are three kinds of Fortran records, as follows:

- Formatted

  A record containing formatted data that requires translation from internal to external form. Formatted I/O statements have explicit format specifiers (which can specify list-directed formatting) or namelist specifiers (for namelist formatting). Only formatted I/O statements can read formatted data.

- Unformatted

  A record containing unformatted data that is not translated from internal form. An unformatted record can also contain no data. The internal representation of unformatted data is processor-dependent. Only unformatted I/O statements can read unformatted data.

- Endfile

The last record of a file. An endfile record can be explicitly written to a sequential file by an ENDFILE statement.

A file is a sequence of records. There are two types of Fortran files, as follows:

- External

  A file that exists in a medium (such as computer disks or terminals) external to the executable program.

  Records in an external file must be either all formatted or all unformatted. There are two ways to access records in external files: sequential and direct access.

  In sequential access, records are processed in the order in which they appear in the file. In direct access, records are selected by record number, so they can be processed in any order.

- Internal

  Memory (internal storage) that behaves like a file. This type of file provides a way to transfer and convert data in memory from one format to another. The contents of these files are stored as scalar character variables.

See Also

- Data Transfer I/O Statements

Building Applications for details on formatted and unformatted data transfers and external file access methods

Components of Data Transfer Statements

Data transfer statements take one of the following forms:

io-keyword (*io-control-list*) [*io-list*]

io-keyword format [, *io-list*]

io-keyword — Is one of the following: ACCEPT, PRINT (or TYPE), READ, REWRITE, or WRITE.

io-control-list — Is one or more of the following input/output (I/O) control specifiers:

[UNIT=]io-unit	ADVANCE	ERR	SIZE
[FMT=]format	END	IOSTAT	
[NML=]group	EOR	REC	

io-list	Is an I/O list, which can contain variables (except for assumed-size arrays) or implied-DO lists. Output statements can contain constants or expressions.
format	Is the nonkeyword form of a control-list format specifier (no FMT=).

If a format specifier ([FMT=]format) or namelist specifier ([NML=]group) is present, the data transfer statement is called a formatted I/O statement; otherwise, it is an unformatted I/O statement.

If a record specifier (REC=) is present, the data transfer statement is a direct-access I/O statement; otherwise, it is a sequential-access I/O statement.

If an error, end-of-record, or end-of-file condition occurs during data transfer, file positioning and execution are affected, and certain control-list specifiers (if present) become defined. (For more information, see Branch Specifiers.)

Following sections describe the I/O control list and I/O lists.

I/O Control List

The I/O control list specifies one or more of the following:

- The I/O unit to act upon ([UNIT=]io-unit)

  This specifier must be present; the rest are optional.
- The format (explicit or list-directed) to use for data editing; if explicit, the keyword form must appear ([FMT=]
- The namelist group name to act upon ([NML=]group)
- The number of a record to access (REC)
- The name of a variable that contains the completion status of an I/O operation (IOSTAT)
- The label of the statement that receives control if an error (ERR), end-of-file (END), or end-of-record (EOR) condition occurs
- Whether you want to use advancing or nonadvancing I/O (ADVANCE)
- The number of characters read from a record (SIZE) by a nonadvancing READ statement
- Whether you want to use asynchronous or synchronous I/O (ASYNCHRONOUS)
- The identifier for a pending data transfer operation (ID)
- The identifier for the file position in file storage units in a stream file (POS)

No control specifier can appear more than once, and the list must not contain both a format specifier and namelist group name specifier.

Control specifiers can take any of the following forms:

- Keyword form

  When the keyword form (for example, UNIT=io-unit) is used for all control-list specifiers in an I/O statement, the specifiers can appear in any order.

- Nonkeyword form

  When the nonkeyword form (for example, io-unit) is used for all control-list specifiers in an I/O statement, the io-unit specifier must be the first item in the control list. If a format specifier or namelist group name specifier is used, it must immediately follow the io-unit specifier.

- Mixed form

  When a mix of keyword and nonkeyword forms is used for control-list specifiers in an I/O statement, the nonkeyword values must appear first. Once a keyword form of a specifier is used, all specifiers to the right must also be keyword forms.

See Also

- Components of Data Transfer Statements
- Unit Specifier
- Format Specifier
- Namelist Specifier
- Record Specifier
- I/O Status Specifier
- Branch Specifiers
- Advance Specifier
- Asynchronous Specifier
- Character Count Specifier
- ID Specifier
- POS Specifier
- Unit Specifier
- Format Specifier
- Namelist Specifier
- Record Specifier
- I/O Status Specifier
- Branch Specifiers
- Advance Specifier
- Asynchronous Specifier
- Character Count Specifier

- ID Specifier
- POS Specifier

Unit Specifier

The unit specifier identifies the I/O unit to be accessed. It takes the following form:

[UNIT=]*io-unit*

io-unit For external files, it identifies a logical unit and is one of the following:

- A scalar integer expression that refers to a specific file, I/O device, or pipe. If necessary, the value is converted to integer data type before use. The integer is in the range 0 through 2,147,483,643. Note that the predefined parameters FOR_K_PRINT_UNITNO, FOR_K_TYPE_UNITNO, FOR_K_ACCEPT_UNITNO, and FOR_K_READ_UNITNO may not be in that range.

  Units 5, 6, and 0 are associated with preconnected units.
- An asterisk (*). This is the default (or implicit) external unit, which is preconnected for formatted sequential access. You can also preconnect files by using an environment variable.

For internal files, *io-unit* identifies a scalar or array character variable that is an internal file. An internal file is designated internal storage space (a variable buffer) that is used with formatted (including list-directed) sequential READ and WRITE statements.

The *io-unit* must be specified in a control list. If the keyword UNIT is omitted, the *io-unit* must be first in the control list.

A unit number is assigned either explicitly through an OPEN statement or implicitly by the system. If a READ statement implicitly opens a file, the file's status is STATUS='OLD'. If a WRITE statement implicitly opens a file, the file's status is STATUS='UNKNOWN'.

If the internal file is a *scalar* character variable, the file has only one record; its length is equal to that of the variable.

If the internal file is an *array* character variable, the file has a record for each element in the array; each record's length is equal to one array element.

An internal file can be read only if the variable has been defined and a value assigned to each record in the file. If the variable representing the internal file is a pointer, it must be associated; if the variable is an allocatable array, it must be currently allocated.

Before data transfer, an internal file is always positioned at the beginning of the first character of the first record.

See Also

- I/O Control List
- OPEN statement

Building Applications for details on implicit logical assignments, details on preconnected units, details on using internal files, and details on using environmental variables

Format Specifier

The format specifier indicates the format to use for data editing. It takes the following form:

[FMT=]*format*

format Is one of the following:

- The statement label of a FORMAT statement

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- An asterisk (*), indicating list-directed formatting
- A scalar default integer variable that has been assigned the label of a FORMAT statement (through an ASSIGN statement)

  The FORMAT statement must be in the same scoping unit as the data transfer statement.
- A character expression (which can be an array or character constant) containing the run-time format

  A default character expression must evaluate to a valid format specification. If the expression is an array, it is treated as if all the elements of the array were specified in array element order and were concatenated.
- The name of a numeric array (or array element) containing the format

If the keyword FMT is omitted, the format specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a format specifier appears in a control list, a namelist group specifier must not appear.

See Also

- I/O Control List
- FORMAT statement
- Interaction between FORMAT statements and I/O lists
- Rules for List-Directed Sequential READ Statements

- Rules for List-Directed Sequential WRITE Statements

Namelist Specifier

The namelist specifier indicates namelist formatting and identifies the namelist group for data transfer. It takes the following form:

[NML=]*group*

group	Is the name of a namelist group previously declared in a NAMELIST statement.

If the keyword NML is omitted, the namelist specifier must be the second specifier in the control list; the io-unit specifier must be first.

If a namelist specifier appears in a control list, a format specifier must *not* appear.

See Also

- I/O Control List
- Rules for Namelist Sequential READ Statements
- Rules for Namelist Sequential WRITE Statements
- READ
- WRITE

Record Specifier

The record specifier identifies the number of the record for data transfer in a file connected for direct access. It takes the following form:

REC=*r*

r	Is a scalar numeric expression indicating the record number. The value of the expression must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file.

If necessary, the value is converted to integer data type before use.

If REC is present, no END specifier, * format specifier, or namelist group name can appear in the same control list.

See Also

- I/O Control List
- Alternative Syntax for a Record Specifier

I/O Status Specifier

The I/O status specifier designates a variable to store a value indicating the status of a data transfer operation. It takes the following form:

IOSTAT=*i-var*

i-var Is a scalar integer variable. When a data transfer statement is executed, *i-var* is set to one of the following values:

A positive integer	Indicating an error condition occurred.
A negative integer	Indicating an end-of-file or end-of-record condition occurred. The negative integers differ depending on which condition occurred.
Zero	Indicating no error, end-of-file, or end-of-record condition occurred.

Execution continues with the statement following the data transfer statement, or the statement identified by a branch specifier (if any).

An end-of-file condition occurs only during execution of a sequential READ statement; an end-of-record condition occurs only during execution of a nonadvancing READ statement.

See Also

- I/O Control List
- CLOSE
- READ
- WRITE

Building Applications for details on the error numbers returned by IOSTAT

Branch Specifiers

A branch specifier identifies a branch target statement that receives control if an error, end-of-file, or end-of-record condition occurs. There are three branch specifiers, taking the following forms:

ERR=*label*

END=*label*

EOR=*label*

label Is the label of the branch target statement that receives control when the specified condition occurs.
The branch target statement must be in the same scoping unit as the data transfer statement.

The following rules apply to these specifiers:

- ERR

  The error specifier can appear in a sequential access READ or WRITE statement, a direct-access READ statement, or a REWRITE statement.

  If an error condition occurs, the position of the file is indeterminate, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a positive integer value. If SIZE was specified (in a nonadvancing READ statement), the SIZE variable becomes defined as an integer value. If a *label* was specified, execution continues with the labeled statement.

- END

  The end-of-file specifier can appear only in a sequential access READ statement.

  An end-of-file condition occurs when no more records exist in a file during a sequential read, or when an end-of-file record produced by the ENDFILE statement is encountered. End-of-file conditions do not occur in direct-access READ statements.

  If an end-of-file condition occurs, the file is positioned after the end-of-file record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If a *label* was specified, execution continues with the labeled statement.

- EOR

  The end-of-record specifier can appear only in a formatted, sequential access READ statement that has the specifier ADVANCE='NO'(nonadvancing input).

  An end-of-record condition occurs when a nonadvancing READ statement tries to transfer data from a position after the end of a record.

  If an end-of-record condition occurs, the file is positioned after the current record, and execution of the statement terminates.

  If IOSTAT was specified, the IOSTAT variable becomes defined as a negative integer value. If PAD='YES' was specified for file connection, the record is padded with blanks (as necessary) to satisfy the input item list and the corresponding data edit descriptor. If SIZE was specified, the SIZE variable becomes defined as an integer value. If a *label* was specified, execution continues with the labeled statement.

If one of the conditions occurs, no branch specifier appears in the control list, but an IOSTAT specifier appears, execution continues with the statement following the I/O statement. If neither a branch specifier nor an IOSTAT specifier appears, the program terminates.

See Also

- I/O Control List
- I/O Status Specifier
- Branch Statements

Building Applications: Using the IOSTAT Specifier and Fortran Exit Codes for more details on the IOSTAT specifier

Building Applications: Using the END, EOR, and ERR Branch Specifiers for more details on branch target statements

Building Applications for details on error processing

Advance Specifier

The advance specifier determines whether nonadvancing I/O occurs for a data transfer statement. It takes the following form:

ADVANCE=*c-expr*

c-expr	Is a scalar character expression that evaluates to 'YES' for advancing I/O or 'NO' for nonadvancing I/O. The default value is 'YES'. Trailing blanks in the expression are ignored. The values specified are without regard to case.

The ADVANCE specifier can appear only in a formatted, sequential data transfer statement that specifies an external unit. It must not be specified for list-directed or namelist data transfer.

Advancing I/O always positions a file at the end of a record, unless an error condition occurs. Nonadvancing I/O can position a file at a character position within the current record.

For details on advancing and nonadvancing I/O, see *Building Applications*.

See Also

- I/O Control List

Building Applications for details on advancing and nonadvancing I/O

Asynchronous Specifier

The asynchronous specifier determines whether asynchronous I/O occurs for a data transfer statement. It takes the following form:

ASYNCHRONOUS=*i-expr*

i-expr	Is a scalar character initialization expression that evaluates to 'YES' for asynchronous I/O or 'NO' for synchronous I/O. The value 'YES' should not appear unless the data transfer statement specifies a file unit number for *io-unit*. The default value is 'NO'.

Trailing blanks in the expression are ignored. The values specified are without regard to case.

Asynchronous I/O is permitted only for external files opened with an OPEN statement that specifies ASYNCHRONOUS='YES'.

When an asynchronous I/O statement is executed, the pending I/O storage sequence for the data transfer operation is defined to be:

- The set of storage units specified by the I/O item list or by the NML= specifier
- The storage units specified by the SIZE= specifier

Character Count Specifier

The character count specifier defines a variable to contain the count of how many characters are read when a nonadvancing READ statement terminates. It takes the following form:

SIZE=*i-var*

i-var Is a scalar integer variable.

If PAD='YES' was specified for file connection, blanks inserted as padding are not counted.

The SIZE specifier can appear only in a formatted, sequential READ statement that has the specifier ADVANCE='NO' (nonadvancing input). It must not be specified for list-directed or namelist data transfer.

ID Specifier

The ID specifier identifies a pending data transfer operation for a specified unit. It takes the following form:

ID=*id-var*

id-var Is a scalar integer variable to be used as an identifier.
This specifier can only be used if the value of ASYNCHRONOUS=*i-expr* is 'YES'.

If an ID specifier is used in a data transfer statement, a wait operation is performed for the operation. If it is omitted, wait operations are performed for all pending data transfers for the specified unit.

If an error occurs during the execution of a data transfer statement containing an ID specifier, the variable specified becomes undefined.

POS Specifier

The POS specifier identifies the file position in file storage units in a stream file (ACCESS='STREAM'). It takes the following form:

POS=*p*

p Is a scalar integer expression that specifies the file position. It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.

Each file storage unit has a unique file position, represented by a positive integer. The first file storage unit is a file is at file position 1. The position of each subsequent file storage unit is one greater than that of its preceding file storage unit.

For a formatted file, the file storage unit is an eight-bit byte. For an unformatted file, the file storage unit is an eight-bit byte (if option assume byterecl is specified) or a 32-bit word (if option assume nobyterecl, the default, is specified).

I/O Lists

In a data transfer statement, the I/O list specifies the entities whose values will be transferred. The I/O list is either an implied-do list or a simple list of variables (except for assumed-size arrays).

In input statements, the I/O list cannot contain constants and expressions because these do not specify named memory locations that can be referenced later in the program.

However, constants and expressions can appear in the I/O lists for output statements because the compiler can use temporary memory locations to hold these values during the execution of the I/O statement.

If an input item is a pointer, it must be currently associated with a definable target; data is transferred from the file to the associated target. If an output item is a pointer, it must be currently associated with a target; data is transferred from the target to the file.

If an input or output item is an array, it is treated as if the elements (if any) were specified in array element order. For example, if ARRAY_A is an array of shape (2,1), the following input statements are equivalent:

```
READ *, ARRAY_A

READ *, ARRAY_A(1,1), ARRAY_A(2,1)
```

However, no element of that array can affect the value of any expression in the input list, nor can any element appear more than once in an input list. For example, the following input statements are invalid:

```
INTEGER  B(50)

...

READ *, B(B)

READ *, B(B(1):B(10))
```

If an input or output item is an allocatable array, it must be currently allocated.

If an input or output item is a derived type, the following rules apply:

- Any derived-type component must be in the scoping unit containing the I/O statement.
- The derived type must not have a pointer component.
- In a formatted I/O statement, a derived type is treated as if all of the components of the structure were specified in the same order as in the derived-type definition.
- In an unformatted I/O statement, a derived type is treated as a single object.

Simple List Items in I/O Lists

In a data transfer statement, a simple list of items takes the following form:

item [, *item*] ...

item — Is one of the following:

- For input statements: a variable name

  The variable must not be an assumed-size array, unless one of the following appears in the last dimension: a subscript, a vector subscript, or a section subscript specifying an upper bound.
- For output statements: a variable name, expression, or constant

  Any expression must not attempt further I/O operations on the same logical unit. For example, it must not refer to a function subprogram that performs I/O on the same logical unit.

The data transfer statement assigns values to (or transfers values from) the list items in the order in which the items appear, from left to right.

When multiple array names are used in the I/O list of an unformatted input or output statement, only one record is read or written, regardless of how many array name references appear in the list.

Examples

The following example shows a simple I/O list:

```
WRITE (6,10) J, K(3), 4, (L+4)/2, N
```

When you use an array name reference in an I/O list, an input statement reads enough data to fill every item of the array. An output statement writes all of the values in the array.

Data transfer begins with the initial item of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. The following statement defines a two-dimensional array:

```
DIMENSION ARRAY(3,3)
```

If the name ARRAY appears with no subscripts in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

An input record contains the following values:

```
1,3,721.73
```

The following example shows how variables in the I/O list can be used in array subscripts later in the list:

```
DIMENSION ARRAY(3,3)

...

READ (1,30) J, K, ARRAY(J,K)
```

When the READ statement is executed, the first input value is assigned to J and the second to K, establishing the subscript values for ARRAY(J,K). The value 721.73 is then assigned to ARRAY(1,3). Note that the variables must appear before their use as array subscripts.

Consider the following derived-type definition and structure declaration:

```
TYPE EMPLOYEE

  INTEGER ID

  CHARACTER(LEN=40) NAME

END TYPE EMPLOYEE

...

TYPE(EMPLOYEE) :: CONTRACT   ! A structure of type EMPLOYEE
```

The following statements are equivalent:

```
READ *, CONTRACT

READ *, CONTRACT%ID, CONTRACT%NAME
```

The following shows more examples:

```
!  A variable and array element in iolist:
      REAL b(99)
      READ (*, 300) n, b(n) ! n and b(n) are the iolist
300   FORMAT (I2, F10.5) ! FORMAT statement telling what form the input data has
!  A derived type and type element in iolist:
      TYPE YOUR_DATA
        REAL a
        CHARACTER(30) info
        COMPLEX cx
      END TYPE YOUR_DATA
      TYPE (YOUR_DATA) yd1, yd2
      yd1.a = 2.3
      yd1.info = "This is a type demo."
      yd1.cx = (3.0, 4.0)
      yd2.cx = (4.5, 6.7)
!  The iolist follows the WRITE (*,500).
      WRITE (*, 500) yd1, yd2.cx
!  The format statement tells how the iolist will be output.
500   FORMAT (F5.3, A21, F5.2, ',', F5.2, ' yd2.cx = (', F5.2,
       ',',F5.2, ' )')
!  The output looks like:
!  2.300This is a type demo 3.00, 4.00 yd2.cx = ( 4.50, 6.70 )
```

The following example uses an array and an array section:

```
!  An array in the iolist:
      INTEGER handle(5)
      DATA handle / 5*0 /
      WRITE (*, 99) handle
99    FORMAT (5I5)
!  An array section in the iolist.
      WRITE (*, 100) handle(2:3)
100   FORMAT (2I5)
```

The following shows another example:

```
PRINT *,'(I5)', 2*3   ! The iolist is the expression 2*3.
```

The following example uses a namelist:

```
! Namelist I/O:
      INTEGER int1
      LOGICAL log1
      REAL r1
      CHARACTER (20) char20
      NAMELIST /mylist/ int1, log1, r1, char20
      int1 = 1
      log1 = .TRUE.
      r1 = 1.0
      char20 = 'NAMELIST demo'
      OPEN (UNIT = 4, FILE = 'MYFILE.DAT', DELIM = 'APOSTROPHE')
      WRITE (UNIT = 4, NML = mylist)
! Writes the following:
! &MYLIST
! INT1 = 1,
! LOG1 = T,
! R1 = 1.000000,
! CHAR20 = 'NAMELIST demo '
! /
      REWIND(4)
      READ (4, mylist)
```

See Also

- I/O Lists
- I/O Lists

Implied-DO Lists in I/O Lists

In a data transfer statement, an implied-DO list acts as though it were a part of an I/O statement within a DO loop. It takes the following form:

(*list*, *do-var* = *expr* 1, *expr* 2 [, *expr* 3])

list	Is a list of variables, expressions, or constants (see Simple List Items in I/O Lists).
do-var	Is the name of a scalar integer or real variable. The variable must not be one of the input items in *list*.
expr	Are scalar numeric expressions of type integer or real. They do not all have to be the same type, or the same type as the DO variable.

The implied-DO loop is initiated, executed, and terminated in the same way as a DO construct.

The *list* is the range of the implied-DO loop. Items in that list can refer to *do-var*, but they must not change the value of *do-var*.

Two nested implied-DO lists must not have the same (or an associated) DO variable.

Use an implied-DO list to do the following:

- Specify iteration of part of an I/O list
- Transfer part of an array
- Transfer array items in a sequence different from the order of subscript progression

If the I/O statement containing an implied-DO list terminates abnormally (with an END, EOR, or ERR branch or with an IOSTAT value other than zero), the DO variable becomes undefined.

Examples

The following two output statements are equivalent:

```
WRITE (3,200) (A,B,C, I=1,3)           ! An implied-DO list

WRITE (3,200) A,B,C,A,B,C,A,B,C        ! A simple item list
```

The following example shows nested implied-DO lists. Execution of the innermost list is repeated most often:

```
WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
```

The inner DO loop is executed 10 times for each iteration of the outer loop; the second subscript (L) advances from 1 through 10 for each increment of the first subscript (K). This is the reverse of the normal array element order. Note that K is incremented by 2, so only the odd-numbered rows of the array are output.

In the following example, the entire list of the implied-DO list (P(1), Q(1,1), Q(1,2)...,Q(1,10)) are read before I is incremented to 2:

```
READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

The following example uses fixed subscripts and subscripts that vary according to the implied-DO list:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

Input values are assigned to BOX(1,1) through BOX(1,10), but other elements of the array are not affected.

The following example shows how a DO variable can be output directly:

```
WRITE (6,1111) (I, I=1,20)
```

Integers 1 through 20 are written.

Consider the following:

```
      INTEGER mydata(25)

      READ (10, 9000) (mydata(I), I=6,10,1)

9000  FORMAT (5I3)
```

In this example, the *iolist* specifies to put the input data into elements 6 through 10 of the array called mydata. The third value in the implied-DO loop, the increment, is optional. If you leave it out, the increment value defaults to 1.

See Also

- I/O Lists
- DO constructs
- I/O Lists

READ Statements Overview

The READ statement is a data transfer input statement. Data can be input from external sequential or direct-access records, or from internal records. For more information, see READ.

See Also

- Data Transfer I/O Statements
- Forms for Sequential READ Statements
- Forms for Direct-Access READ Statements
- Forms for Stream READ Statements
- Forms and Rules for Internal READ Statements
- Forms for Sequential READ Statements
- Forms for Direct-Access READ Statements

- Forms for STREAM READ Statements
- Forms and Rules for Internal READ Statements

Forms for Sequential READ Statements

Sequential READ statements transfer input data from external sequential-access records. The statements can be formatted with format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential READ statement takes one of the following forms:

Formatted:

READ (*eunit*, *format* [, *advance*] [, *asynchronous*] [, *id*] [, *pos*] [, *size*] [, *iostat*] [, *err*] [, *end*] [, *eor*]) [*io-list*]

READ *form* [, *io-list*]

Formatted - List-Directed:

READ (*eunit*, * [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *end*]) [*io-list*]

READ * [, *io-list*]

Formatted - Namelist:

READ (*eunit*, *nml-group* [, *iostat*] [, *err*] [, *end*])

READ *nml*

Unformatted:

READ (*eunit* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*] [, *end*]) [*io-list*]

See Also

- READ Statements Overview
- Rules for Formatted Sequential READ Statements
- Rules for List-Directed Sequential READ Statements
- Rules for Namelist Sequential READ Statement
- Rules for Unformatted Sequential READ Statements
- READ

- Rules for Formatted Sequential READ Statements
- Rules for List-Directed Sequential READ Statements
- Rules for Namelist Sequential READ Statements
- Rules for Unformatted Sequential READ Statements
- I/O control-list specifiers
- I/O lists

Rules for Formatted Sequential READ Statements

Formatted, sequential READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential READ statements:

```
READ (*, '(B)', ADVANCE='NO') C

READ (FMT="(E2.4)", UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

See Also

- Forms for Sequential READ Statements
- READ statement
- Forms for Sequential READ Statements

Rules for List-Directed Sequential READ Statements

List-directed, sequential READ statements translate data from character to binary form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then assigned to the entities in the I/O list in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

List-Directed Records

A list-directed external record consists of a sequence of values and value separators. A value can be any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see the table in Numeric Assignment Statements).
  - A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.
  - A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

  A character string does not need delimiting apostrophes or quotation marks if the corresponding I/O list item is of type default character, and the following is true:

  - The character string does not contain a blank, comma (,), or slash (/).
  - The character string is not continued across a record boundary.

- The first nonblank character in the string is not an apostrophe or a quotation mark.
- The leading character is not a string of digits followed by an asterisk.

A nondelimited character string is terminated by the first blank, comma, slash, or end-of-record encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.

- A null value

  A null value is specified by two consecutive value separators (such as,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

  A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Examples

Suppose the following statements are specified:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
```

Then suppose the following external record is read:

```
4 6.3 (3.4,4.2), (3, 2 ), T,F,,3*14.6,'ABC,DEF/GHI''JK'/
```

The following values are assigned to the I/O list items:

I/O List Item	Value Assigned
I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
J	Unchanged
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI' JK
A	Unchanged
B	Unchanged

The following example shows list-directed input and output:

```
REAL    a
INTEGER i
COMPLEX c
LOGICAL up, down
DATA a /2358.2E-8/, i /91585/, c /(705.60,819.60)/
DATA up /.TRUE./, down /.FALSE./
OPEN (UNIT = 9, FILE = 'listout', STATUS = 'NEW')
WRITE  (9, *) a, i
WRITE  (9, *) c, up, down
REWIND (9)
READ   (9, *) a, i
READ   (9, *) c, up, down
WRITE  (*, *) a, i
WRITE  (*, *) c, up, down
END
```

The preceding program produces the following output:

```
  2.3582001E-05     91585
 (705.6000,819.6000) T F
```

See Also

- Forms for Sequential READ Statements
- READ
- Forms for Sequential READ Statements
- Intrinsic Data Types
- Rules for List-Directed Sequential WRITE Statements

Rules for Namelist Sequential READ Statement

Namelist, sequential READ statements translate data from external to internal form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is assigned to the specified objects in the namelist group in the order in which they appear, from left to right.

If a slash (/) is encountered during execution, the READ statement is terminated, and any remaining input list items are unchanged.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Namelist Records

A namelist external record takes the following form:

&*group-name object* = *value* [, *object* = *value*] .../

group-name Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit. The name cannot contain embedded blanks and must be contained within a single record.

object Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks except within the parentheses of a subscript or substring specifier. Each object must be contained in a single record.

value Is any of the following:

- A constant

  Each constant must be a literal constant of type integer, real, complex, logical, or character; or a nondelimited character string. Binary, octal, hexadecimal, Hollerith, and named constants are not permitted.

  In general, the form of the constant must be acceptable for the type of the list item. The data type of the constant determines the data type of the value and the translation from external to internal form. The following rules also apply:

  - A numeric list item can correspond only to a numeric constant, and a character list item can correspond only to a character constant. If the data types of a numeric list element and its corresponding numeric constant do not match, conversion is performed according to the rules for arithmetic assignment (see the table in Numeric Assignment Statements). Logical list items and logical constants are not considered numeric, unless the compiler option `assume old_logical_ldio` is specified.

- A complex constant has the form of a pair of real or integer constants separated by a comma and enclosed in parentheses. Blanks can appear between the opening parenthesis and the first constant, before and after the separating comma, and between the second constant and the closing parenthesis.
- A logical constant represents true values (.TRUE. or any value beginning with T, .T, t, or .t) or false values (.FALSE. or any value beginning with F, .F, f, or .f).

A character string does not need delimiting apostrophes or quotation marks if the corresponding NAMELIST item is of type default character, and the following is true:

- The character string does not contain a blank, comma (,), slash (/), exclamation point(!), ampersand (&), dollar sign ($), left parenthesis, equal sign (=), percent sign (%), or period (.).

  The character string is not continued across a record boundary.
- The first nonblank character in the string is not an apostrophe or a quotation mark.
- The leading characters are not a string of digits followed by an asterisk.

A nondelimited character string is terminated by the first blank, comma, slash, end-of-record, exclamation, ampersand, or dollar sign encountered. Apostrophes and quotation marks within nondelimited character strings are transferred as is.

If an equal sign, percent sign, or period is encountered while scanning for a nondelimited character string, the string is treated as a variable name (or part of one) and not as a nondelimited character string.

- A null value

  A null value is specified by two consecutive value separators (such as,,) or a nonblank initial value separator. (A value separator before the end of the record does not signify a null value.)

  A null value indicates that the corresponding list element remains unchanged. A null value can represent an entire complex constant, but cannot be used for either part of a complex constant.

- A repetition of a null value (r*) or a constant (r*constant), where *r* is an unsigned, nonzero, integer literal constant with no kind parameter, and no embedded blanks.

Blanks can precede or follow the beginning ampersand (&), follow the group name, precede or follow the equal sign, or precede the terminating slash.

Comments (beginning with ! only) can appear anywhere in namelist input. The comment extends to the end of the source line.

If an entity appears more than once within the input record for a namelist data transfer, the last value is the one that is used.

If there is more than one *object* = *value* pair, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks. When any of these appear in a character constant, they are considered part of the constant, not value separators.

The end of a record is equivalent to a blank character, except when it occurs in a character constant. In this case, the end of the record is ignored, and the character constant is continued with the next record (the last character in the previous record is immediately followed by the first character of the next record).

Blanks at the beginning of a record are ignored unless they are part of a character constant continued from the previous record. In this case, the blanks at the beginning of the record are considered part of the constant.

Prompting for Namelist Group Information

During execution of a program containing a namelist READ statement, you can specify a question mark character (?) or a question mark character preceded by an equal sign (=?) to get information about the namelist group. The ? or =? must follow one or more blanks.

If specified for a unit capable of both input and output, the ? causes display of the group name and the objects in that group. The =? causes display of the group name, objects within that group, and the current values for those objects (in namelist output form). If specified for another type of unit, the symbols are ignored.

For example, consider the following statements:

```
NAMELIST /NLIST/ A,B,C
REAL A /1.5/
INTEGER B /2/
CHARACTER*5 C /'ABCDE'/
READ (5,NML=NLIST)
WRITE (6,NML=NLIST)
END
```

During execution, if a blank followed by ? is entered on a terminal device, the following values are displayed:

```
&NLIST
   A
   B
   C
/
```

If a blank followed by =? is entered, the following values are displayed:

```
&NLIST
    A  =   1.500000,
    B  =           2,
    C  = ABCDE
/
```

Examples

Suppose the following statements are specified:

```
NAMELIST /CONTROL/ TITLE, RESET, START, STOP, INTERVAL
CHARACTER*10 TITLE
REAL(KIND=8) START, STOP
LOGICAL(KIND=4) RESET
INTEGER(KIND=4) INTERVAL
READ (UNIT=1, NML=CONTROL)
```

The NAMELIST statement associates the group name CONTROL with a list of five objects. The corresponding READ statement reads the following input data from unit 1:

```
 &CONTROL
        TITLE='TESTT002AA',
        INTERVAL=1,
        RESET=.TRUE.,
        START=10.2,
        STOP =14.5
/
```

The following values are assigned to objects in group CONTROL:

Namelist Object	Value Assigned
TITLE	TESTT002AA
RESET	T
START	10.2
STOP	14.5
INTERVAL	1

It is not necessary to assign values to all of the objects declared in the corresponding NAMELIST group. If a namelist object does not appear in the input statement, its value (if any) is unchanged.

Similarly, when character substrings and array elements are specified, only the values of the specified variable substrings and array elements are changed. For example, suppose the following input is read:

```
&CONTROL TITLE(9:10)='BB' /
```

The new value for TITLE is TESTT002BB; only the last two characters in the variable change.

The following example shows an array as an object:

```
 DIMENSION ARRAY_A(20)
 NAMELIST /ELEM/ ARRAY_A
 READ (UNIT=1,NML=ELEM)
```

Suppose the following input is read:

```
&ELEM
ARRAY_A=1.1, 1.2,, 1.4
/
```

The following values are assigned to the ARRAY_A elements:

Array Element	Value Assigned
ARRAY_A(1)	1.1
ARRAY_A(2)	1.2
ARRAY_A(3)	Unchanged
ARRAY_A(4)	1.4
ARRAY_A(5)...ARRAY(20)	Unchanged

When a list of values is assigned to an array element, the assignment begins with the specified array element, rather than with the first element of the array. For example, suppose the following input is read:

```
&ELEM
ARRAY_A(3)=34.54, 45.34, 87.63, 3*20.00
/
```

New values are assigned only to array ARRAY_A elements 3 through 8. The other element values are unchanged.

The following shows another example:

```
        INTEGER a, b
        NAMELIST /mynml/ a, b
 ...
 !  The following are all valid namelist variable assignments:
        &mynml a = 1 /
        $mynml a = 1 $
        $mynml a = 1 $end
        &mynml a = 1 &
        &mynml a = 1 $END
        &mynml
        a = 1
        b = 2
        /
```

Nondelimited character strings that are written out by using a NAMELIST write may not be read in as expected by a corresponding NAMELIST read. Consider the following:

```
NAMELIST/TEST/ CHARR
CHARACTER*3 CHARR(4)
DATA CHARR/'AAA', 'BBB', 'CCC', 'DDD'/
OPEN (UNIT=1, FILE='NMLTEST.DAT')
WRITE (1, NML=TEST)
END
```

The output file NMLTEST.DAT will contain:

```
&TESTCHARR   = AAABBBCCCDDD/
```

If an attempt is then made to read the data in NMLTEST.DAT with a NAMELIST read using nondelimited character strings, as follows:

```
NAMELIST/TEST/ CHARR

CHARACTER*3 CHARR(4)

DATA CHARR/4*'   '/

OPEN (UNIT=1, FILE='NMLTEST.DAT')

READ (1, NML=TEST)

PRINT *, 'CHARR read in >', CHARR(1),'<  >',CHARR(2),'<  >',

1       CHARR(3), '<  >', CHARR(4), '<'

END
```

The result is the following:

```
CHARR read in >AAA<  >   <  >   <  >   <
```

See Also

- Forms for Sequential READ Statements
- NAMELIST
- Rules for Formatted Sequential READ Statements
- Alternative Form for Namelist External Records
- Rules for Namelist Sequential WRITE Statements

Rules for Unformatted Sequential READ Statements

Unformatted, sequential READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, sequential READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is *greater* than the number of fields in an input record, an error occurs.

If a statement contains no I/O list, it skips over one full record, positioning the file to read the following record on the next execution of a READ statement.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows an unformatted, sequential READ statement:

```
READ (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

See Also

- Forms for Sequential READ Statements
- READ statement
- Forms for Sequential READ Statements

Forms for Direct-Access READ Statements

Direct-access READ statements transfer input data from external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access READ statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

READ (*eunit*, *format*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

Unformatted:

READ (*eunit*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

See Also

- READ Statements Overview
- Rules for Formatted Direct-Access READ Statements
- Rules for Unformatted Direct-Access READ Statements
- READ
- Rules for Formatted Direct-Access READ Statements
- Rules for Unformatted Direct-Access READ Statements
- I/O control-list specifiers
- I/O lists

Building Applications for details on file sharing

Rules for Formatted Direct-Access READ Statements

Formatted, direct-access READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

For data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains. If more characters are required and nonadvancing input is in effect, an end-of-record condition occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is read by that input statement.

Examples

The following example shows a formatted, direct-access READ statement:

```
READ (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access READ Statements

Unformatted, direct-access READ statements transfer binary data (without translation) between the current record and the entities specified in the I/O list. Only one record is read.

Objects of intrinsic or derived types can be transferred.

For data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

The unformatted, direct-access READ statement reads a single record. Each value in the record must be of the same type as the corresponding entity in the input list, unless the value is real or complex.

If the value is real or complex, one complex value can correspond to two real list entities, or two real values can correspond to one complex list entity. The corresponding values and entities must have the same kind parameter.

If the number of I/O list items is less than the number of fields in an input record, the statement ignores the excess fields. If the number of I/O list items is greater than the number of fields in an input record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access READ statements:

```
READ (1, REC=10) LIST(1), LIST(8)

READ (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms for Stream READ Statements

The forms for stream READ statements take the same forms as sequential READ statements. A POS specifier may be present to specify at what file position the READ will start.

You can impose a record structure on a formatted, sequential stream by using a new-line character as a record terminator (see intrinsic function NEW_LINE). There is no record structure in an unformatted, sequential stream.

The INQUIRE statement can be used with the POS specifier to determine the current file position in a stream file.

Examples

The following example shows stream READ statements:

```
READ (12 ) I         !stream reading without POS= specifier

READ (12,POS=10) J   !stream reading with POS= specifier
```

See Also

- READ Statements Overview
- NEW_LINE

Forms and Rules for Internal READ Statements

Internal READ statements transfer input data from an internal file.

An internal READ statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is not permitted.

An internal READ statement takes the following form:

READ (*iunit*, *format* [, *iostat*] [, *err*] [, *end*]) [*io-list*]

For more information on syntax, see READ.

Formatted, internal READ statements translate data from character to binary form by using format specifications for editing (if any). The translated data is assigned to the entities in the I/O list in the order in which the entities appear, from left to right.

This form of READ statement behaves as if the format begins with a BN edit descriptor. (You can override this behavior by explicitly specifying the BZ edit descriptor.)

Values can be transferred to objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred to the components of intrinsic types that ultimately make up these structured objects.

Before data transfer occurs, the file is positioned at the beginning of the first record. This record becomes the current record.

If the number of I/O list items is *less* than the number of fields in an input record, the statement ignores the excess fields.

If the number of I/O list items is *greater* than the number of fields in an input record, the input record is padded with blanks. However, if PAD='NO' was specified for file connection, the input list and file specification must not require more characters from the record than it contains.

In list-directed formatting, character strings have no delimiters.

Examples

The following program segment reads a record and examines the first character to determine whether the remaining data should be interpreted as decimal, octal, or hexadecimal. It then uses internal READ statements to make appropriate conversions from character string representations to binary.

```
INTEGER IVAL
CHARACTER TYPE, RECORD*80
CHARACTER*(*) AFMT, IFMT, OFMT, ZFMT
PARAMETER (AFMT='(Q,A)', IFMT= '(I10)', OFMT= '(O11)',        &  ZFMT= '(Z8)')
ACCEPT AFMT, ILEN, RECORD
TYPE = RECORD(1:1)
IF (TYPE .EQ. 'D') THEN
    READ (RECORD(2:MIN(ILEN, 11)), IFMT) IVAL
ELSE IF (TYPE .EQ. 'O') THEN
    READ (RECORD(2:MIN(ILEN, 12)), OFMT) IVAL
ELSE IF (TYPE .EQ. 'X') THEN
    READ (RECORD(2:MIN(ILEN, 9)),ZFMT) IVAL
ELSE
    PRINT *, 'ERROR'
END IF
END
```

See Also

- READ Statements Overview
- I/O control-list specifiers
- I/O lists
- Rules for List-Directed Sequential READ Statements

Building Applications for details on using internal files

ACCEPT Statement Overview

The ACCEPT statement is a data transfer input statement. This statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units. You can override this restriction by using an environment variable. For more information, see *Building Applications*: Logical Devices.

WRITE Statements Overview

The WRITE statement is a data transfer output statement. Data can be output to external sequential or direct-access records, or to internal records. For more information, see WRITE.

See Also

- Data Transfer I/O Statements
- Forms for Sequential WRITE Statements
- Forms for Direct-Access WRITE Statements
- Forms for Stream WRITE Statements
- Forms and Rules for Internal WRITE Statements
- Forms for Sequential WRITE Statements
- Forms for Direct-Access WRITE Statements
- Forms for STREAM WRITE Statements
- Forms and Rules for Internal WRITE Statements

Forms for Sequential WRITE Statements

Sequential WRITE statements transfer output data to external sequential access records. The statements can be formatted by using format specifiers (which can use list-directed formatting) or namelist specifiers (for namelist formatting), or they can be unformatted.

A sequential WRITE statement takes one of the following forms:

Formatted:

WRITE (*eunit*, *format* [, *advance*] [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

Formatted - List-Directed:

WRITE (*eunit*, * [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

Formatted - Namelist:

WRITE (*eunit*, *nml-group* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*])

Unformatted:

WRITE (*eunit* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

See Also

- WRITE Statements Overview
- Rules for Formatted Sequential WRITE Statements
- Rules for List-Directed Sequential WRITE Statements
- Rules for Namelist Sequential WRITE Statements
- Rules for Unformatted Sequential WRITE Statements
- WRITE
- Rules for Formatted Sequential WRITE Statements
- Rules for List-Directed Sequential WRITE Statements
- Rules for Namelist Sequential WRITE Statements
- Rules for Unformatted Sequential WRITE Statements
- I/O control-list specifiers
- I/O lists

Rules for Formatted Sequential WRITE Statements

Formatted, sequential WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for sequential access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

The output list and format specification must not specify more characters for a record than the record size. (Record size is specified by RECL in an OPEN statement.)

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

Examples

The following example shows formatted, sequential WRITE statements:

```
WRITE (UNIT=8, FMT='(B)', ADVANCE='NO') C

WRITE (*, "(F6.5)", ERR=25, IOSTAT=IO_STATUS) A, B, C
```

See Also

- Forms for Sequential WRITE Statements
- WRITE statement
- Forms for Sequential WRITE Statements

Rules for List-Directed Sequential WRITE Statements

List-directed, sequential WRITE statements transfer data from binary to character form by using the data types of the corresponding I/O list item to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

The following table shows the default output formats for each intrinsic data type:

Table 610: Default Formats for List-Directed Output

Data Type	Output Format
BYTE	I5
LOGICAL(1)	L2
LOGICAL(2)	L2
LOGICAL(4)	L2
LOGICAL(8)	L2
INTEGER(1)	I5
INTEGER(2)	I7
INTEGER(4)	I12
INTEGER(8)	I22
REAL(4)	1PG15.7E2
REAL(8)	1PG24.15E3
REAL(16)	1PG43.33E4
COMPLEX(4)	'(',1PG14.7E2,',',1PG14.7E2,')'
COMPLEX(8)	'(',1PG23.15E3,',',1PG23.15E3,')'

Data Type	Output Format
COMPLEX(16)	'(',1PG42.33E4,',',1PG42.33E4,')'
CHARACTER	Aw [1]

[1] Where w is the length of the character expression.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. For complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record. For literal constants that are longer than an entire record, the constant is continued onto as many records as necessary.

Each output record begins with a blank character for carriage control.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, list-directed data transfer is prohibited.

Examples

Suppose the following statements are specified:

```
DIMENSION A(4)
DATA A/4*3.4/
WRITE (1,*) 'ARRAY VALUES FOLLOW'
WRITE (1,*) A,4
```

The following records are then written to external unit 1:

```
ARRAY VALUES FOLLOW

   3.400000      3.400000      3.400000      3.400000      4
```

The following shows another example:

```
  INTEGER       i, j

  REAL          a, b

  LOGICAL       on, off

  CHARACTER(20) c

  DATA i /123456/, j /500/, a /28.22/, b /.0015555/

  DATA on /.TRUE./, off/.FALSE./

  DATA c /'Here''s a string'/

  WRITE (*, *)  i, j

  WRITE (*, *)  a, b, on, off

  WRITE (*, *)  c

  END
```

The preceding example produces the following output:

```
     123456         500

  28.22000      1.555500E-03 T F

Here's a string
```

See Also

- Forms for Sequential WRITE Statements
- Rules for Formatted Sequential WRITE Statements
- Rules for List-Directed Sequential READ Statements

Rules for Namelist Sequential WRITE Statements

Namelist, sequential WRITE statements translate data from internal to external form by using the data types of the objects in the corresponding NAMELIST statement to determine the form of the data. The translated data is then written to an external file.

In general, values transferred as output have the same forms as values transferred as input.

By default, character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

This behavior can be changed by the DELIM specifier (in an OPEN statement) as follows:

- If the file is opened with the DELIM='QUOTE' specifier, character constants are delimited by quotation marks and each internal quotation mark is represented externally by two consecutive quotation marks.
- If the file is opened with the DELIM='APOSTROPHE' specifier, character constants are delimited by apostrophes and each internal apostrophe is represented externally by two consecutive apostrophes.

Each output statement writes one or more complete records.

A literal character constant or complex constant can be longer than an entire record. In the case of complex constants, the end of the record can occur between the comma and the imaginary part, if the imaginary part and closing right parenthesis cannot fit in the current record.

Each output record begins with a blank character for carriage control, except for literal character constants that are continued from the previous record.

Slashes, octal values, null values, and repeated forms of values are not output.

If the file is connected for unformatted I/O, namelist data transfer is prohibited.

Examples

Consider the following statements:

```
CHARACTER*19 NAME(2)/2*' '/

REAL PITCH, ROLL, YAW, POSITION(3)

LOGICAL DIAGNOSTICS

INTEGER ITERATIONS

NAMELIST /PARAM/ NAME, PITCH, ROLL, YAW, POSITION,        &

         DIAGNOSTICS, ITERATIONS

... READ (UNIT=1,NML=PARAM)

WRITE (UNIT=2,NML=PARAM)
```

Suppose the following input is read:

```
&PARAM
    NAME(2)(10:)='HEISENBERG',
    PITCH=5.0, YAW=0.0, ROLL=5.0,
    DIAGNOSTICS=.TRUE.
    ITERATIONS=10
/
```

The following is then written to the file connected to unit 2:

```
&PARAM
NAME   = '                          ', '        HEISENBERG',
PITCH  =    5.000000,
ROLL   =    5.000000,
YAW    =   0.0000000E+00,
POSITION        = 3*0.0000000E+00,
DIAGNOSTICS     = T,
ITERATIONS      =                10
/
```

Note that character values are not enclosed in apostrophes unless the output file is opened with DELIM='APOSTROPHE'. The value of POSITION is not defined in the namelist input, so the current value of POSITION is written.

The following example declares a number of variables, which are placed in a namelist, initialized, and then written to the screen with namelist I/O:

```
      INTEGER(1) int1
      INTEGER    int2, int3, array(3)
      LOGICAL(1) log1
      LOGICAL log2, log3
      REAL    real1
      REAL(8) real2
      COMPLEX z1, z2
      CHARACTER(1)  char1
      CHARACTER(10) char2
      NAMELIST /example/ int1, int2, int3, log1, log2, log3,       &
     &          real1, real2, z1, z2, char1, char2, array
      int1     = 11
      int2     = 12
      int3     = 14
      log1     = .TRUE.
      log2     = .TRUE.
      log3     = .TRUE.
      real1    = 24.0
      real2    = 28.0d0
      z1       = (38.0,0.0)
      z2       = (316.0d0,0.0d0)
      char1    = 'A'
      char2    = '0123456789'
      array(1) = 41
      array(2) = 42
      array(3) = 43
      WRITE (*, example)
```

The preceding example produces the following output:

```
&EXAMPLE
INT1  =   11,
INT2  =          12,
INT3  =          14,
LOG1  = T,
LOG2  = T,
LOG3  = T,
REAL1 =   24.00000,
REAL2 =   28.0000000000000,
Z1    = (38.00000,0.0000000E+00),
Z2    = (316.0000,0.0000000E+00),
CHAR1 = A,
CHAR2 = 0123456789,
ARRAY =          41,          42,          43
/
```

See Also

- Forms for Sequential WRITE Statements
- NAMELIST
- Rules for Formatted Sequential WRITE Statements
- Rules for Namelist Sequential READ Statements

Rules for Unformatted Sequential WRITE Statements

Unformatted, sequential WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

This form of WRITE statement writes exactly one record. If there is no I/O item list, the statement writes one null record.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows an unformatted, sequential WRITE statement:

```
WRITE (UNIT=6, IOSTAT=IO_STATUS) A, B, C
```

Forms for Direct-Access WRITE Statements

Direct-access WRITE statements transfer output data to external records with direct access. (The attributes of a direct-access file are established by the OPEN statement.)

A direct-access WRITE statement can be formatted or unformatted, and takes one of the following forms:

Formatted:

WRITE (*eunit*, *format*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

Unformatted:

WRITE (*eunit*, *rec* [, *asynchronous*] [, *id*] [, *pos*] [, *iostat*] [, *err*]) [*io-list*]

See Also

- WRITE Statements Overview
- Rules for Formatted Direct-Access WRITE Statements
- Rules for Unformatted Direct-Access WRITE Statements
- WRITE
- Rules for Formatted Direct-Access WRITE Statements
- Rules for Unformatted Direct-Access WRITE Statements
- I/O control-list specifiers
- I/O lists

Rules for Formatted Direct-Access WRITE Statements

Formatted, direct-access WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an external file that is connected for direct access.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the format specification specifies another record, the record number is increased by one as each subsequent record is written by that output statement.

Examples

The following example shows a formatted, direct-access WRITE statement:

```
WRITE (2, REC=35, FMT=10) (NUM(K), K=1,10)
```

Rules for Unformatted Direct-Access WRITE Statements

Unformatted, direct-access WRITE statements transfer binary data (without translation) between the entities specified in the I/O list and the current record. Only one record is written.

Objects of intrinsic or derived types can be transferred.

If the values specified by the I/O list do not fill a record, blank characters are added to fill the record. If the I/O list specifies too many characters for the record, an error occurs.

If the file is connected for formatted, list-directed, or namelist I/O, unformatted data transfer is prohibited.

Examples

The following example shows unformatted, direct-access WRITE statements:

```
WRITE (1, REC=10) LIST(1), LIST(8)

WRITE (4, REC=58, IOSTAT=K, ERR=500) (RHO(N), N=1,5)
```

Forms for Stream WRITE Statements

The forms for stream WRITE statements take the same forms as sequential WRITE statements. A POS specifier may be present to specify at what file position the WRITE will start.

You can impose a record structure on a formatted, sequential stream by using a new-line character as a record terminator (see intrinsic function NEW_LINE). There is no record structure in an unformatted, sequential stream.

The INQUIRE statement can be used with the POS specifier to determine the current file position in a stream file.

See Also

- WRITE Statements Overview
- NEW_LINE

Forms and Rules for Internal WRITE Statements

Internal WRITE statements transfer output data to an internal file.

An internal WRITE statement can only be formatted. It must include format specifiers (which can use list-directed formatting). Namelist formatting is not permitted.

An internal WRITE statement takes the following form:

WRITE (*iunit*, *format* [, *iostat*] [, *err*]) [*io-list*]

For more information on syntax, see WRITE.

Formatted, internal WRITE statements translate data from binary to character form by using format specifications for editing (if any). The translated data is written to an internal file.

Values can be transferred from objects of intrinsic or derived types. For derived types, values of intrinsic types are transferred from the components of intrinsic types that ultimately make up these structured objects.

If the number of characters written in a record is less than the length of the record, the rest of the record is filled with blanks. The number of characters to be written must not exceed the length of the record.

Character constants are not delimited by apostrophes or quotation marks, and each internal apostrophe or quotation mark is represented externally by one apostrophe or quotation mark.

Examples

The following example shows an internal WRITE statement:

```
INTEGER J, K, STAT_VALUE

CHARACTER*50 CHAR_50

...

WRITE (FMT=*, UNIT=CHAR_50, IOSTAT=STAT_VALUE) J, K
```

See Also

- WRITE Statements Overview
- I/O control-list specifiers
- I/O lists
- Rules for List-Directed Sequential WRITE Statements

Building Applications for details on using internal files

PRINT and TYPE Statements Overview

The PRINT statement is a data transfer output statement. TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.

The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units. You can override this restriction by using an environment variable. For more information, see *Building Applications*: Logical Devices.

REWRITE Statement Overview

The REWRITE statement is a data transfer output statement that rewrites the current record.

A REWRITE statement can be formatted or unformatted. For more information, see REWRITE in the *A to Z Reference*.

I/O Formatting

52

A format appearing in an input or output (I/O) statement specifies the form of data being transferred and the data conversion (editing) required to achieve that form. The format specified can be explicit or implicit.

Explicit format is indicated in a format specification that appears in a FORMAT statement or a character expression (the expression must evaluate to a valid format specification).

The format specification contains edit descriptors, which can be data edit descriptors, control edit descriptors, or string edit descriptors.

Implicit format is determined by the processor and is specified using list-directed or namelist formatting.

List-directed formatting is specified with an asterisk (*); namelist formatting is specified with a namelist group name.

List-directed formatting can be specified for advancing sequential files and internal files. Namelist formatting can be specified only for advancing sequential files.

This chapter contains information on the following topics:

- Format specifications
- Data edit descriptors
- Control edit descriptors
- Character string edit descriptors
- Nested and group repeat specifications
- Variable Format Expressions
- Printing of formatted records
- Interaction between FORMAT statements and I/O lists

Format Specifications

A format specification can appear in a FORMAT statement or character expression. In a FORMAT statement, it is preceded by the keyword FORMAT. A format specification takes the following form:

(*format-list*)

format-list Is a list of one or more of the following edit descriptors, separated by commas or slashes (/):

Data edit descriptors:	I, B, O, Z, F, E, EN, ES, D, G, L, and A

Control edit descriptors:	T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, $, \, and Q
String edit descriptors:	H, 'c', and "c", where *c* is a character constant

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Edit descriptors can be nested and a *repeat specification* can precede data edit descriptors, the slash edit descriptor, or a parenthesized list of edit descriptors.

Description

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see BLANK Specifier in OPEN statements.)

For formatted input, use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a blank, 0, 1, $, +, or ASCII NUL. Any other character is treated as a blank.

A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

The following table summarizes the edit descriptors that can be used in format specifications.

Table 612: Summary of Edit Descriptors

Code	Form	Effect
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
BN	BN	Ignores embedded and trailing blanks in a numeric input field.
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
D	Dw.d	Transfers real values with D exponents.
E	Ew.d[Ee]	Transfers real values with E exponents.
EN	ENw.d[Ee]	Transfers real values with engineering notation.
ES	ESw.d[Ee]	Transfers real values with scientific notation.
F	Fw.d	Transfers real values with no exponent.

Code	Form	Effect
G	Gw.d[Ee]	Transfers values of all intrinsic types.
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.

Code	Form	Effect
TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
Z	Zw[.m]	Transfers hexadecimal values.
$	$	Suppresses trailing carriage return during interactive I/O.
:	:	Terminates format control if there are no more items in the I/O list.
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.
'c' [1]	'c'	Transfers the character literal constant (between the delimiters) to an output record.

[1] These delimiters can also be quotation marks (").

Character Format Specifications

In data transfer I/O statements, a format specifier ([FMT=]format) can be a character expression that is a character array, character array element, or character constant. This type of format is also called a run-time format because it can be constructed or altered during program execution.

The expression must evaluate to a character string whose leading part is a valid format specification (including the enclosing parentheses).

If the expression is a character array element, the format specification must be contained entirely within that element.

If the expression is a character array, the format specification can continue past the first element into subsequent consecutive elements.

If the expression is a character constant delimited by apostrophes, use two consecutive apostrophes ('') to represent an apostrophe character in the format specification; for example:

```
PRINT '("NUM can''t be a real number")'
```

Similarly, if the expression is a character constant delimited by quotation marks, use two consecutive quotation marks ("") to represent a quotation mark character in the format specification.

To avoid using consecutive apostrophes or quotation marks, you can put the character constant in an I/O list instead of a format specification, as follows:

```
PRINT "(A)", "NUM can't be a real number"
```

The following shows another character format specification:

```
WRITE (6, '(I12, I4, I12)') I, J, K
```

In the following example, the format specification changes with each iteration of the DO loop:

```
SUBROUTINE PRINT(TABLE)
REAL TABLE(10,5)
CHARACTER*5 FORCHR(0:5), RPAR*1, FBIG, FMED, FSML
DATA FORCHR(0),RPAR /'(',')'/
DATA FBIG,FMED,FSML /'F8.2,','F9.4,','F9.6,'/
DO I=1,10
  DO J=1,5
    IF (TABLE(I,J) .GE. 100.) THEN
      FORCHR(J) = FBIG
    ELSE IF (TABLE(I,J) .GT. 0.1) THEN
      FORCHR(J) = FMED
    ELSE
      FORCHR(J) = FSML
    END IF
  END DO
  FORCHR(5)(5:5) = RPAR
  WRITE (6,FORCHR) (TABLE(I,J), J=1,5)
END DO
END
```

The DATA statement assigns a left parenthesis to character array element FORCHR(0), and (for later use) a right parenthesis and three F edit descriptors to character variables.

Next, the proper F edit descriptors are selected for inclusion in the format specification. The selection is based on the magnitude of the individual elements of array TABLE.

A right parenthesis is added to the format specification just before the WRITE statement uses it.

NOTE. Format specifications stored in arrays are recompiled at run time each time they are used. If a Hollerith or character run-time format is used in a READ statement to read data into the format itself, that data is not copied back into the original array, and the array is unavailable for subsequent use as a run-time format specification.

Examples

The following example shows a format specification:

```
        WRITE (*, 9000) int1, real1(3), char1
9000    FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 is the format list.
```

In the following example, the integer-variable name MYFMT refers to the FORMAT statement 9000, as assigned just before the FORMAT statement.

```
      ASSIGN 9000 TO MYFMT
9000  FORMAT (I5, 3F4.5, A16)
!  I5, 3F5.2, A16 is the format list.
      WRITE (*, MYFMT) iolist
```

The following shows a format example using a character expression:

```
      WRITE (*, '(I5, 3F5.2, A16)')iolist
! I5, 3F4.5, A16 is the format list.
```

In the following example, the format list is put into an 80-character variable called MYLIST:

```
CHARACTER(80) MYLIST
MYLIST = '(I5, 3F5.2, A16)'
WRITE (*, MYLIST) iolist
```

Consider the following two-dimensional array:

```
   1  2  3
   4  5  6
```

In this case, the elements are stored in memory in the order: 1, 4, 2, 5, 3, 6 as follows:

```
CHARACTER(6) array(3)
DATA array / '(I5', ',3F5.2', ',A16)' /
WRITE (*, array) iolist
```

In the following example, the WRITE statement uses the character array element array(2) as the format specifier for data transfer:

```
CHARACTER(80) array(5)

array(2) = '(I5, 3F5.2, A16)'

WRITE (*, array(2)) iolist
```

See Also

- I/O Formatting
- Data edit descriptors
- Control edit descriptors
- Character string edit descriptors
- Nested and group repeats
- Printing of formatted records

Data Edit Descriptors

A data edit descriptor causes the transfer or conversion of data to or from its internal representation.

The part of a record that is input or output and formatted with data edit descriptors (or character string edit descriptors) is called a *field*.

See Also

- I/O Formatting
- Forms for Data Edit Descriptors
- General Rules for Numeric Editing
- Integer Editing
- Real and Complex Editing
- Logical Editing (L)
- Character Editing (A)
- Default Widths for Data Edit Descriptors
- Terminating Short Fields of Input Data
- Forms for Data Edit Descriptors
- General Rules for Numeric Editing
- Integer Editing
- Real and Complex Editing

- Logical Editing (L)
- Character Editing (A)
- Default Widths for Data Edit Descriptors
- Terminating Short Fields of Input Data

Forms for Data Edit Descriptors

A data edit descriptor takes one of the following forms:

[r]c

[r]cw

[r]cw.m

[r]cw.d

[r]cw.d[Ee]

r	Is a repeat specification. The range of *r* is 1 through 2147483647 (2**31-1). If *r* is omitted, it is assumed to be 1.
c	Is one of the following format codes: I, B, O, Z, F, E, EN, ES, D, G, L, or A.
w	Is the total number of digits in the field (the field width). If omitted, the system applies default values (see Default Widths for Data Edit Descriptors). The range of *w* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture and IA-64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. For I, B, O, Z, and F, the range can start at zero.
m	Is the minimum number of digits that must be in the field (including leading zeros). The range of *m* is 0 through 32767 (2**15-1) on Intel® 64 architecture and IA-64 architecture; 0 through 255 (2**8-1) on IA-32 architecture.
d	Is the number of digits to the right of the decimal point (the significant digits). The range of *d* is 0 through 32767 (2**15-1) on Intel® 64 architecture and IA-64 architecture; 0 through 255 (2**8-1) on IA-32 architecture. The number of significant digits is affected if a scale factor is specified for the data edit descriptor.
E	Identifies an exponent field.
e	Is the number of digits in the exponent. The range of *e* is 1 through 32767 (2**15-1) on Intel® 64 architecture and IA-64 architecture; 1 through 255 (2**8-1) on IA-32 architecture.

Description

Fortran 95/90 (and the previous standard) allows the field width to be omitted only for the A descriptor. However, Intel® Fortran allows the field width to be omitted for any data edit descriptor.

The *r*, *w*, *m*, *d*, and *e* must all be positive, unsigned, integer literal constants; or variable format expressions -- no kind parameter can be specified. They must not be named constants.

Actual useful ranges for *r*, *w*, *m*, *d*, and *e* may be constrained by record sizes (RECL) and the file system.

The data edit descriptors have the following specific forms:

Integer:	Iw[.m], Bw[.m], Ow[.m], and Zw[.m]
Real and complex:	Fw.d, Ew.d[Ee], ENw.d[Ee], ESw.d[Ee], Dw.d, and Gw.d[Ee]
Logical:	Lw
Character:	A[w]

The *d* must be specified with F, E, D, and G field descriptors even if *d* is zero. The decimal point is also required. You must specify both *w* and *d*, or omit them both.

A repeat specification can simplify formatting. For example, the following two statements are equivalent:

```
20  FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
20  FORMAT (3E12.4,4I5)
```

Examples

```
!  This WRITE outputs three integers, each in a five-space field
!  and four reals in pairs of F7.2 and F5.2 values.
      INTEGER(2) int1, int2, int3
      REAL(4) r1, r2, r3, r4
      DATA int1, int2, int3 /143, 62, 999/
      DATA r1, r2, r3, r4 /2458.32, 43.78, 664.55, 73.8/
      WRITE (*,9000) int1, int2, int3, r1, r2, r3, r4
9000  FORMAT (3I5, 2(1X, F7.2, 1X, F5.2))
```

The following output is produced:

```
  143   62   999 2458.32 43.78  664.55 73.80
```

See Also

- Data Edit Descriptors
- General rules for numeric editing
- Nested and group repeats

General Rules for Numeric Editing

The following rules apply to input and output data for numeric editing (data edit descriptors I, B, O, Z, F, E, EN, ES, D, and G).

Rules for Input Processing

Leading blanks in the external field are ignored. If BLANK='NULL' is in effect (or the BN edit descriptor has been specified) embedded and trailing blanks are ignored; otherwise, they are treated as zeros. An all-blank field is treated as a value of zero.

The following table shows how blanks are interpreted by default:

Type of Unit or File	Default
An explicitly OPENed unit	BLANK='NULL'
An internal file	BLANK='NULL'
A preconnected file[1]	BLANK='NULL'

Type of Unit or File	Default

[1] For interactive input from preconnected files, you should explicitly specify the BN or BZ edit descriptor to ensure desired behavior.

A minus sign must precede a negative value in an external field; a plus sign is optional before a positive value.

In input records, constants can include any valid kind parameter. Named constants are not permitted.

If the data field in a record contains fewer than *w* characters, an input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data. The comma terminates the data field, and can also be used to designate null (zero-length) fields. For more information, see Terminating Short Fields of Input Data.

Rules for Output Processing

The field width *w* must be large enough to include any leading plus or minus sign, and any decimal point or exponent. For example, the field width for an E data edit descriptor must be large enough to contain the following:

- For positive numbers: *d* + 5 or *d* + *e* + 3 characters
- For negative numbers: *d* + 6 or *d* + *e* + 4 characters

A positive or zero value (zero is allowed for I, B, O, Z, and F descriptors) can have a plus sign, depending on which sign edit descriptor is in effect. If a value is negative, the leftmost nonblank character is a minus sign.

If the value is smaller than the field width specified, leading blanks are inserted (the value is right-justified). If the value is too large for the field width specified, the entire output field is filled with asterisks (*).

When the value of the field width is zero, the compiler selects the smallest possible positive actual field width that does not result in the field being filled with asterisks.

See Also

- Data Edit Descriptors
- Forms for data edit descriptors
- Format Specifications

Compiler Options for details on compiler options

Integer Editing

Integer editing is controlled by the I (decimal), B (binary), O (octal), and Z (hexadecimal) data edit descriptors.

I Editing

The I edit descriptor transfers decimal integer values. It takes the following form:

I*w*[.*m*]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item must be of type integer or logical.

The G edit descriptor can be used to edit integer data; it follows the same rules as I*w*.

Rules for Input Processing

On input, the I data edit descriptor transfers *w* characters from an external field and assigns their integer value to the corresponding I/O list item. The external field data must be an integer constant.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the I edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Input	Value
I4	2788	2788
I3	-26	-26
I9	^^^^^^312	312

Rules for Output Processing

On output, the I data edit descriptor transfers the value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by a sign (a plus sign is optional for positive values, a minus sign is required for negative values), followed by an unsigned integer constant with no leading zeros.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the I edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
I3	284	284
I4	-284	-284
I4	0	^^^0
I5	174	^^174
I2	3244	**
I3	-473	***
I7	29.812	An error; the decimal point is invalid
I4.0	0	^^^^
I4.2	1	^^01
I4.4	1	0001

See Also

- Integer Editing
- Forms for data edit descriptors
- General rules for numeric editing

B Editing

The B data edit descriptor transfers binary (base 2) values. It takes the following form:

Bw[.m]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the B data edit descriptor transfers *w* characters from an external field and assigns their binary value to the corresponding I/O list item. The external field must contain only binary digits (0 or 1) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the B edit descriptor:

Format	Input	Value
B4	1001	9
B1	1	1
B2	0	0

Rules for Output Processing

On output, the B data edit descriptor transfers the binary value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of binary digits) with no leading zeros. A negative value is transferred in internal form.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the B edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
B4	9	1001
B2	0	^0

See Also

- Integer Editing
- Forms for data edit descriptors
- General rules for numeric editing

O Editing

The O data edit descriptor transfers octal (base 8) values. It takes the following form:

Ow[.m]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the O data edit descriptor transfers *w* characters from an external field and assigns their octal value to the corresponding I/O list item. The external field must contain only octal digits (0 through 7) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the O edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Input	Value
O5	32767	32767
O4	16234	1623
O3	97^	An error; the 9 is invalid in octal notation

Rules for Output Processing

On output, the O data edit descriptor transfers the octal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of octal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the O edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
O6	32767	^77777
O12	-32767	^37777700001
O2	14261	**
O4	27	^^33
O5	10.5	41050
O4.2	7	^^07
O4.4	7	0007

See Also

- Integer Editing
- Forms for data edit descriptors
- General rules for numeric editing

Z Editing

The Z data edit descriptor transfers hexadecimal (base 16) values. It takes the following form:

Zw[.m]

The value of *m* (the minimum number of digits in the constant) must not exceed the value of *w* (the field width). The *m* has no effect on input, only output.

The specified I/O list item can be of type integer, real, or logical.

Rules for Input Processing

On input, the Z data edit descriptor transfers *w* characters from an external field and assigns their hexadecimal value to the corresponding I/O list item. The external field must contain only hexadecimal digits (0 though 9 and A (a) through F(f)) or blanks.

If the value exceeds the range of the corresponding input list item, an error occurs.

The following shows input using the Z edit descriptor:

```
Format     Input          Value
Z3         A94                    A94
Z5         A23DEF               A23DE
Z5         95.AF2         An error; the decimal point is invalid
```

Rules for Output Processing

On output, the Z data edit descriptor transfers the hexadecimal value of the corresponding I/O list item, right-justified, to an external field that is *w* characters long.

The field consists of zero or more blanks, followed by an unsigned integer constant (consisting of hexadecimal digits) with no leading zeros. A negative value is transferred in internal form without a leading minus sign.

If *m* is specified, the unsigned integer constant must have at least *m* digits. If necessary, it is padded with leading zeros.

If *m* is zero, and the output list item has the value zero, the external field is filled with blanks.

The following shows output using the Z edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format       Value          Output
Z4            32767         7FFF
Z9           -32767         ^FFFF8001
Z2               16         10
Z4            -10.5         ****
Z3.3           2708         A94
Z6.4           2708         ^^0A94
```

See Also

- Integer Editing
- Forms for data edit descriptors
- General rules for numeric editing

Real and Complex Editing

Real and complex editing is controlled by the F, E, D, EN, ES, and G data edit descriptors.

If no field width (*w*) is specified for a real data edit descriptor, the system supplies default values.

Real data edit descriptors can be affected by specified scale factors.

NOTE. Do not use the real data edit descriptors when attempting to parse textual input. These descriptors accept some forms that are purely textual as valid numeric input values. For example, input values T and F are treated as values -1.0 and 0.0, respectively, for .TRUE. and .FALSE.

F Editing

The F data edit descriptor transfers real values. It takes the following form:

Fw.d

The value of *d* (the number of places after the decimal point) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the F data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The external field data must be an integer or real constant.

If the input field contains only an exponent letter or decimal point, it is treated as a zero value.

If the input field does not contain a decimal point or an exponent, it is treated as a real number of *w* digits, with *d* digits to the right of the decimal point. (Leading zeros are added, if necessary.)

If the input field contains a decimal point, the location of that decimal point overrides the location specified by the F descriptor.

If the field contains an exponent, that exponent is used to establish the magnitude of the value before it is assigned to the list element.

The following shows input using the F edit descriptor:

Format	Input	Value
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

Rules for Output Processing

On output, the F data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to d decimal positions, to an external field that is w characters long.

The w must be greater than or equal to d+3 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- At least one digit to the left of the decimal point
- The decimal point
- The d digits to the right of the decimal point

The following shows output using the F edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
F8.5	2.3547188	^2.35472
F9.3	8789.7361	^8789.736
F2.1	51.44	**
F10.4	-23.24352	^^-23.2435
F5.2	325.013	******
F5.2	-.2	-0.20

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing

E and D Editing

The E and D data edit descriptors transfer real values in exponential form. They take the following form:

Ew.d[Ee]

Dw.d

For the E edit descriptor, the value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

For the D edit descriptor, the value of *d* must not exceed the value of *w*.

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the E and D data edit descriptors transfer *w* characters from an external field and assigns their real value to the corresponding I/O list item. The E and D descriptors interpret and assign input data in the same way as the F data edit descriptor.

The following shows input using the E and D edit descriptors (the symbol ^ represents a nonprinting blank character):

Format	Input	Value
E9.3	734.432E3	734432.0
E12.4	^^1022.43E	1022.43E-6
E15.3	52.3759663^^^^^	52.3759663
E12.5	210.5271D+10	210.5271E10
BZ,D10.2	12345^^^^^	12345000.0D0
D10.2	^^123.45^^	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

If the I/O list item is single-precision real, the E edit descriptor treats the D exponent indicator as an E indicator.

Rules for Output Processing

On output, the E and D data edit descriptors transfer the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- An optional zero to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
Ew.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
Ew.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$
Dw.d	|exp| 99	D+nn or E+nn	D-nn or E-nn
	99 < |exp| 999	+nnn	-nnn

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional for the E edit descriptor; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d*+*e*+5.

NOTE. The *w* can be as small as *d* + 5 or *d* + *e* + 3, if the optional fields for the sign and the zero are omitted.

The following shows output using the E and D edit descriptors (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
E11.2	475867.222	^^^0.48E+06
E11.5	475867.222	0.47587E+06
E12.3	0.00069	^^^0.690E
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****
E14.5E4	-1.001	-0.10010E+0001
E13.3E6	0.000123	0.123E-000003
D14.3	0.0363	^^^^^0.363D-01
D23.12	5413.87625793	^^^^^0.541387625793D+04
D9.6	1.2	*********

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing
- Scale Factor Editing (P)

EN Editing

The EN data edit descriptor transfers values by using engineering notation. It takes the following form:

ENw.d[Ee]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the EN data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The EN descriptor interprets and assigns input data in the same way as the F data edit descriptor.

The following shows input using the EN edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Input           Value
EN11.3       ^^5.321E+00       5.32100
EN11.3       -600.00E-03       -.60000
EN12.3      ^^^3.150E-03        .00315
EN12.3      ^^^3.829E+03    3829.0
```

Rules for Output Processing

On output, the EN data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in engineering notation, where the decimal exponent is divisible by 3 and the absolute value of the significand is greater than or equal to 1 and less than 1000 (unless the output value is zero).

The *w* should be greater than or equal to *d*+9 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One to three digits to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ENw.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
ENw.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d* + *e* + 5.

The following shows output using the EN edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
EN11.2	475867.222	^475.87E+03
EN11.5	475867.222	**********
EN12.3	0.00069	^690.000E-06
EN10.3	-0.5555	**********
EN11.2	0.0	^000.00E-03

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing

ES Editing

The ES data edit descriptor transfers values by using scientific notation. It takes the following form:

ESw.d[Ee]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item must be of type real, or it must be the real or imaginary part of a complex type.

Rules for Input Processing

On input, the ES data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The ES descriptor interprets and assigns input data in the same way as the F data edit descriptor.

The following shows input using the ES edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Input           Value
ES11.3       ^^5.321E+00       5.32100
ES11.3        -6.000E-03       -.60000
ES12.3      ^^^3.150E-03        .00315
ES12.3      ^^^3.829E+03    3829.0
```

Rules for Output Processing

On output, the ES data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long. The real value is output in scientific notation, where the absolute value of the significand is greater than or equal to 1 and less than 10 (unless the output value is zero).

The *w* should be greater than or equal to *d*+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The *d* digits to the right of the decimal point
- The exponent

The exponent takes one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Positive Form of Exponent	Negative Form of Exponent
ESw.d	|exp| 99	E+nn	E-nn
	99 < |exp| 999	+nnn	-nnn
ESw.dEe	|exp| 10^e - 1	E+$n_1n_2...n_e$	E-$n_1n_2...n_e$

If the exponent value is too large to be converted into one of these forms, an error occurs.

The exponent field width (*e*) is optional; if omitted, the default value is 2. If *e* is specified, the *w* should be greater than or equal to *d* + *e* + 5.

The following shows output using the ES edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
ES11.2	473214.356	^^^4.73E+05
ES11.5	473214.356	4.73214E+05
ES12.3	0.00069	^^^6.900E-04
ES10.3	-0.5555	-5.555E-01
ES11.2	0.0	^0.000E+00

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing

G Editing

The G data edit descriptor generally transfers values of real type, but it can be used to transfer values of any intrinsic type. It takes the following form:

Gw.d[Ee]

The value of *d* (the number of places after the decimal point) plus *e* (the number of digits in the exponent) must not exceed the value of *w* (the field width).

The specified I/O list item can be of any intrinsic type.

When used to specify I/O for integer, logical, or character data, the edit descriptor follows the same rules as I*w*, L*w*, and A*w*, respectively, and *d* and *e* have no effect.

Rules for Real Input Processing

On input, the G data edit descriptor transfers *w* characters from an external field and assigns their real value to the corresponding I/O list item. The G descriptor interprets and assigns input data in the same way as the F data edit descriptor.

Rules for Real Output Processing

On output, the G data edit descriptor transfers the real value of the corresponding I/O list item, right-justified and rounded to *d* decimal positions, to an external field that is *w* characters long.

The form in which the value is written is a function of the magnitude of the value, as described in the following table:

Table 618: Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
$0 < m < 0.1 - 0.5 \times 10^{-d-1}$	Ew.d[Ee]
$m = 0$	F(w - n).(d -1), n('b')
$0.1 - 0.5 \times 10^{-d-1}$ $m < 1 - 0.5 \times 10^{-d}$	F(w - n).d, n('b')
$1 - 0.5 \times 10^{-d}$ $m < 10 - 0.5 \times 10^{-d+1}$	F(w - n).(d -1), n('b')
$10 - 0.5 \times 10^{-d+1}$ $m < 100 - 0.5 \times 10^{-d+2}$	F(w - n).(d -2), n('b')
.	.
.	.
.	.
$10^{d-2} - 0.5 \times 10^{-2}$ $m < 10^{d-1} - 0.5 \times 10^{-1}$	F(w - n).1, n('b')
$10^{d-1} - 0.5 \times 10^{-1}$ $m < 10^{d} - 0.5$	(w - n).0, n('b')
$m\ 10^{d} - 0.5$	Ew.d[Ee]

The 'b' is a blank following the numeric data representation. For Gw.d, n('b') is 4 blanks. For Gw.dEe, n('b') is e+2 blanks.

The w should be greater than or equal to d+7 to allow for the following:

- A sign (optional if the value is positive and descriptor SP is not in effect)
- One digit to the left of the decimal point
- The decimal point
- The d digits to the right of the decimal point
- The 4-digit or e+2-digit exponent

If e is specified, the w should be greater than or equal to $d + e + 5$.

The following shows output using the G edit descriptor and compares it to output using equivalent F editing (the symbol ^ represents a nonprinting blank character):

Value	Format	Output with G	Format	Output with F
0.01234567	G13.6	^0.123457E-01	F13.6	^^^^^0.012346
-0.12345678	G13.6	-0.123457^^^^	F13.6	^^^^-0.123457
1.23456789	G13.6	^^1.23457^^^^	F13.6	^^^^^1.234568
12.34567890	G13.6	^^12.3457^^^^	F13.6	^^^^12.345679
123.45678901	G13.6	^^123.457^^^^	F13.6	^^^123.456789
-1234.56789012	G13.6	^-1234.57^^^^	F13.6	^-1234.567890
12345.67890123	G13.6	^^12345.7^^^^	F13.6	^12345.678901
123456.78901234	G13.6	^^123457.^^^^	F13.6	123456.789012
-1234567.89012345	G13.6	-0.123457E+07	F13.6	**********

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing
- I data edit descriptor
- L data edit descriptor
- A data edit descriptor
- Scale Factor Editing (P)

Complex Editing

A complex value is an ordered pair of real values. Complex editing is specified by a pair of real edit descriptors, using any combination of the forms: Fw.d, Ew.d[Ee], Dw.d, ENw.d[Ee], ESw.d[Ee], or Gw.d[Ee].

Rules for Input Processing

On input, the two successive fields are read and assigned to the corresponding complex I/O list item as its real and imaginary part, respectively.

The following shows input using complex editing:

Format	Input	Value
F8.5,F8.5	1234567812345.67	123.45678, 12345.67
E9.1,F9.3	734.432E8123456789	734.432E8, 123456.789

Rules for Output Processing

On output, the two parts of the complex value are transferred under the control of repeated or successive real edit descriptors. The two parts are transferred consecutively without punctuation or blanks, unless control or character string edit descriptors are specified between the pair of real edit descriptors.

The following shows output using complex editing (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
2F8.5	2.3547188, 3.456732	^2.35472 ^3.45673
E9.2,'^,^',E5.3	47587.222, 56.123	^0.48E+06^,^*****

See Also

- Real and Complex Editing
- Forms for data edit descriptors
- General rules for numeric editing
- General Rules for Complex Constants

Logical Editing (L)

The L data edit descriptor transfers logical values. It takes the following form:

Lw

The specified I/O list item must be of type logical or integer.

The G edit descriptor can be used to edit logical data; it follows the same rules as L*w*.

Rules for Input Processing

On input, the L data edit descriptor transfers *w* characters from an external field and assigns their logical value to the corresponding I/O list item. The value assigned depends on the external field data, as follows:

- .TRUE. is assigned if the first nonblank character is .T, T, .t, or t. The logical constant .TRUE. is an acceptable input form.
- .FALSE. is assigned if the first nonblank character is .F, F. .f, or f, or the entire field is filled with blanks. The logical constant .FALSE. is an acceptable input form.

If an other value appears in the external field, an error occurs.

Rules for Output Processing

On output, the L data edit descriptor transfers the following to an external field that is *w* characters long: w - 1 blanks, followed by a T or F (if the value is .TRUE. or .FALSE., respectively).

The following shows output using the L edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
L5	.TRUE.	^^^^T
L1	.FALSE.	F

See Also

- Data Edit Descriptors
- Forms for data edit descriptors

Character Editing (A)

The A data edit descriptor transfers character or Hollerith values. It takes the following form:

A[w]

If the corresponding I/O list item is of type character, character data is transferred. If the list item is of any other type, Hollerith data is transferred.

The G edit descriptor can be used to edit character data; it follows the same rules as A*w*.

Rules for Input Processing

On input, the A data edit descriptor transfers *w* characters from an external field and assigns them to the corresponding I/O list item.

The maximum number of characters that can be stored depends on the size of the I/O list item, as follows:

- For character data, the maximum size is the length of the corresponding I/O list item.
- For noncharacter data, the maximum size depends on the data type, as shown in the following table:

Table 619: Size Limits for Noncharacter Data Using A Editing

I/O List Element	Maximum Number of Characters
BYTE	1
LOGICAL(1) or LOGICAL*1	1
LOGICAL(2) or LOGICAL*2	2
LOGICAL(4) or LOGICAL*4	4
LOGICAL(8) or LOGICAL*8	8
INTEGER(1) or INTEGER*1	1
INTEGER(2) or INTEGER*2	2
INTEGER(4) or INTEGER*4	4
INTEGER(8) or INTEGER*8	8
REAL(4) or REAL*4	4
DOUBLE PRECISION	8
REAL(8) or REAL*8	8
REAL(16) or REAL*16	16
COMPLEX(4) or COMPLEX*8[1]	8
DOUBLE COMPLEX[1]	16
COMPLEX(8) or COMPLEX*16[1]	16
COMPLEX(16) or COMPLEX*32[1]	32

I/O List Element	Maximum Number of Characters
[1] Complex values are treated as pairs of real numbers, so complex editing requires a pair of edit descriptors. (See Complex Editing.)	

If *w* is equal to or greater than the length (*len*) of the input item, the rightmost characters are assigned to that item. The leftmost excess characters are ignored.

If *w* is less than *len*, or less than the number of characters that can be stored, *w* characters are assigned to the list item, left-justified, and followed by trailing blanks.

The following shows input using the A edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Input	Value	Data Type
A6	PAGE^#	#	CHARACTER(LEN=1)
A6	PAGE^#	E^#	CHARACTER(LEN=3)
A6	PAGE^#	PAGE^#	CHARACTER(LEN=6)
A6	PAGE^#	PAGE^#^^	CHARACTER(LEN=8)
A6	PAGE^#	#	LOGICAL(1)
A6	PAGE^#	^#	INTEGER(2)
A6	PAGE^#	GE^#	REAL(4)
A6	PAGE^#	PAGE^#^^	REAL(8)

Rules for Output Processing

On output, the A data edit descriptor transfers the contents of the corresponding I/O list item to an external field that is *w* characters long.

If *w* is greater than the size of the list item, the data is transferred to the output field, right-justified, with leading blanks. If *w* is less than or equal to the size of the list item, the leftmost *w* characters are transferred.

The following shows output using the A edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format      Value          Output
A5          OHMS           ^OHMS
A5          VOLTS          VOLTS
A5          AMPERES        AMPER
```

See Also

- Data Edit Descriptors
- Forms for data edit descriptors

Default Widths for Data Edit Descriptors

If *w* (the field width) is omitted for the data edit descriptors, the system applies default values. For the real data edit descriptors, the system also applies default values for *d* (the number of characters to the right of the decimal point), and *e* (the number of characters in the exponent).

These defaults are based on the data type of the I/O list item, and are listed in the following table:

Table 620: Default Widths for Data Edit Descriptors

Edit Descriptor	Data Type of I/O List Item	w
I, B, O, Z, G	BYTE	7
	INTEGER(1), LOGICAL(1)	7
	INTEGER(2), LOGICAL(2)	7
	INTEGER(4), LOGICAL(4)	12
	INTEGER(8), LOGICAL(8)	23
O, Z	REAL(4)	12
	REAL(8)	23
	REAL(16)	44
	CHARACTER*len	MAX(7, 3*len)

Edit Descriptor	Data Type of I/O List Item	w
L, G	LOGICAL(1), LOGICAL(2), LOGICAL(4), LOGICAL(8)	2
F, E, EN, ES, G, D	REAL(4), COMPLEX(4)	15 **d**: 7 **e**: 2
	REAL(8), COMPLEX(8)	25 **d**: 16 **e**: 2
	REAL(16), COMPLEX(16)	42 **d**: 33 **e**: 3
A[1], G	LOGICAL(1)	1
	LOGICAL(2), INTEGER(2)	2
	LOGICAL(4), INTEGER(4)	4
	LOGICAL(8), INTEGER(8)	8
	REAL(4), COMPLEX(4)	4
	REAL(8), COMPLEX(8)	8
	REAL(16), COMPLEX(16)	16
	CHARACTER*len	len

[1] The default is the actual length of the corresponding I/O list item.

Terminating Short Fields of Input Data

On input, an edit descriptor such as Fw.d specifies that *w* characters (the field width) are to be read from the external field.

If the field contains fewer than *w* characters, the input statement will read characters from the next data field in the record. You can prevent this by padding the short field with blanks or zeros, or by using commas to separate the input data.

Padding Short Fields

You can use the OPEN statement specifier PAD='YES' to indicate blank padding for short fields of input data. However, blanks can be interpreted as blanks *or* zeros, depending on which default behavior is in effect at the time. Consider the following:

```
READ (2, '(I5)') J
```

If 3 is input for J, the value of J will be 30000 or 3 depending on which default behavior is in effect (BLANK='NULL' or BLANK='ZERO'). This can give unexpected results.

To ensure that the desired behavior is in effect, explicitly specify the BN or BZ edit descriptor. For example, the following ensures that blanks are interpreted as blanks (and not as zeros):

```
READ (2, '(BN, I5)') J
```

Using Commas to Separate Input Data

You can use a comma to terminate a short data field. The comma has no effect on the *d* part (the number of characters to the right of the decimal point) of the specification.

The comma overrides the *w* specified for the I, B, O, Z, F, E, D, EN, ES, G, and L edit descriptors. For example, suppose the following statements are executed:

```
      READ (5,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

Suppose a record containing the following values is read:

```
1, -2, 1.0, 35
```

The following assignments occur:

```
I = 1
J = -2
A = 1.0
B = 0.35
```

A comma can only terminate fields less than *w* characters long. If a comma follows a field of *w* or more characters, the comma is considered part of the next field.

A null (zero-length) field is designated by two successive commas, or by a comma after a field of *w* characters. Depending on the field descriptor specified, the resulting value assigned is 0, 0.0, 0.0D0, 0.0Q0, or .FALSE..

See Also

- Data Edit Descriptors
- General Rules for Numeric Editing

Control Edit Descriptors

A control edit descriptor either directly determines how text is displayed or affects the conversions performed by subsequent data edit descriptors.

See Also

- I/O Formatting
- Forms for Control Edit Descriptors
- Positional Editing
- Sign Editing
- Blank Editing
- Scale-Factor Editing (P)
- Slash Editing (/)
- Colon Editing (:)
- Dollar-Sign ($) and Backslash (\ ) Editing
- Character Count Editing (Q)
- Forms for Control Edit Descriptors
- Positional Editing
- Sign Editing
- Blank Editing
- Scale Factor Editing (P)
- Slash Editing (/)
- Colon Editing (:)
- Dollar Sign ($) and Backslash (\) Editing
- Character Count Editing (Q)

Forms for Control Edit Descriptors

A control edit descriptor takes one of the following forms:

c

cn

nc

c Is one of the following format codes: T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, \, $, and Q.

n Is a number of character positions. It must be a positive integer literal constant or a variable format expression. No kind parameter can be specified. It cannot be a named constant.
The range of *n* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture and IA-64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. Actual useful ranges may be constrained by record sizes (RECL) and the file system.

Description

In general, control edit descriptors are nonrepeatable. The only exception is the slash (/) edit descriptor, which can be preceded by a repeat specification.

The control edit descriptors have the following specific forms:

Positional:	Tn, TLn, TRn, and nX
Sign:	S, SP, and SS
Blank interpretation:	BN and BZ
Scale factor:	kP
Miscellaneous:	:, /, \, $, and Q

The P edit descriptor is an exception to the general control edit descriptor syntax. It is preceded by a scale factor, rather than a character position specifier.

Control edit descriptors can be grouped in parentheses and preceded by a group repeat specification.

See Also

- Control Edit Descriptors
- Group repeat specifications
- Format Specifications

Positional Editing

The T, TL, TR, and X edit descriptors specify the position where the next character is transferred to or from a record.

On output, these descriptors do not themselves cause characters to be transferred and do not affect the length of the record. If characters are transferred to positions at or after the position specified by one of these descriptors, positions skipped and not previously filled are filled with blanks. The result is as if the entire record was initially filled with blanks.

The TR and X edit descriptors produce the same results.

T Editing

The T edit descriptor specifies a character position in an I/O record. It takes the following form:

Tn

The *n* is a positive integer literal constant (with no kind parameter) indicating the character position of the record, relative to the left tab limit.

On input, the T descriptor positions the external record at the character position specified by *n*. On output, the T descriptor indicates that data transfer begins at the *n*th character position of the external record.

Examples

In the following examples, the symbol ^ represents a nonprinting blank character.

Suppose a file has a record containing the value ABC^^^XYZ, and the following statements are executed:

```
      READ (11,10) VALUE1, VALUE2
10    FORMAT (T7,A3,T1,A3)
```

The values read first are XYZ, then ABC.

Suppose the following statements are executed:

```
     PRINT 25
25   FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

The following line is printed at the positions indicated:

```
Position 20                  Position 50
   |                            |
   COLUMN 1                     COLUMN 2
```

Note that the first character of the record printed was reserved as a control character.

See Also

- Positional Editing
- Printing of Formatted Records

TL Editing

The TL edit descriptor specifies a character position to the *left* of the current position in an I/O record. It takes the following form:

TLn

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the left of the current character.

If *n* is greater than or equal to the current position, the next character accessed is the first character of the record.

TR Editing

The TR edit descriptor specifies a character position to the *right* of the current position in an I/O record. It takes the following form:

TRn

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

X Editing

The X edit descriptor specifies a character position to the right of the current position in an I/O record. It takes the following form:

nX

The *n* is a positive integer literal constant (with no kind parameter) indicating the *n*th character position to the right of the current character.

On output, the X edit descriptor does not output any characters when it appears at the end of a format specification; for example:

```
     WRITE (6,99) K

99   FORMAT ('^K=',I6,5X)
```

Note that the symbol ^ represents a nonprinting blank character. This example writes a record of only 9 characters. To cause *n* trailing blanks to be output at the end of a record, specify a format of `n('^')`.

Sign Editing

The S, SP, and SS edit descriptors control the output of the optional plus (+) sign within numeric output fields. These descriptors have no effect during execution of input statements.

Within a format specification, a sign editing descriptor affects all subsequent I, F, E, EN, ES, D, and G descriptors until another sign editing descriptor occurs.

Examples

```
      INTEGER i
      REAL r
!     The following statements write:
!     251 +251 251 +251 251
      i = 251
      WRITE (*, 100) i, i, i, i, i
100   FORMAT (I5, SP, I5, SS, I5, SP, I5, S, I5)
!     The following statements write:
!     0.673E+4 +.673E+40.673E+4 +.673E+40.673E+4
      r = 67.3E2
      WRITE (*, 200) r, r, r, r, r
200   FORMAT (E8.3E1, 1X, SP, E8.3E1, SS, E8.3E1, 1X, SP, &
     &            E8.3E1, S, E8.3E1)
```

SP Editing

The SP edit descriptor causes the processor to *produce* a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SP

SS Editing

The SS edit descriptor causes the processor to *suppress* a plus sign in any subsequent position where it would be otherwise optional. It takes the following form:

SS

S Editing

The S edit descriptor restores the plus sign as optional for all subsequent positive numeric fields. It takes the following form:

S

The S edit descriptor restores to the processor the discretion of producing plus characters on an optional basis.

Blank Editing

The BN and BZ descriptors control the interpretation of embedded and trailing blanks within numeric input fields. These descriptors have no effect during execution of output statements.

Within a format specification, a blank editing descriptor affects all subsequent I, B, O, Z, F, E, EN, ES, D, and G descriptors until another blank editing descriptor occurs.

The blank editing descriptors override the effect of the BLANK specifier during execution of a particular input data transfer statement. (For more information, see the BLANK specifier in OPEN statements.)

BN Editing

The BN edit descriptor causes the processor to *ignore* all embedded and trailing blanks in numeric input fields. It takes the following form:

BN

The input field is treated as if all blanks have been removed and the remainder of the field is right-justified. An all-blank field is treated as zero.

Examples

If an input field formatted as a six-digit integer (I6) contains '2 3 4', it is interpreted as ' 234'.

Consider the following code:

```
      READ (*, 100) n
100   FORMAT (BN, I6)
```

If you enter any one of the following three records and terminate by pressing Enter, the READ statement interprets that record as the value 123:

```
   123
123
123   456
```

Because the repeatable edit descriptor associated with the I/O list item n is I6, only the first six characters of each record are read (three blanks followed by 123 for the first record, and 123 followed by three blanks for the last two records). Because blanks are ignored, all three records are interpreted as 123.

The following example shows the effect of BN editing with an input record that has fewer characters than the number of characters specified by the edit descriptors and *iolist*. Suppose you enter 123 and press Enter in response to the following READ statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. With BN editing in effect, the nonblank characters (123) are right-aligned, so the record is equal to 123.

BZ Editing

The BZ edit descriptor causes the processor to *interpret* all embedded and trailing blanks in numeric input fields as zeros. It takes the following form:

BZ

Examples

The input field ' 23 4 ' would be interpreted as ' 23040'. If ' 23 4' were entered, the formatter would add one blank to pad the input to the six-digit integer format (I6), but this extra space would be ignored, and the input would be interpreted as ' 2304 '. The blanks following the E or D in real-number input are ignored, regardless of the form of blank interpretation in effect.

Suppose you enter 123 and press Enter in response to the following READ statement:

```
READ (*, '(I6)') n
```

The I/O system is looking for six characters to interpret as an integer number. You have entered only three, so the first thing the I/O system does is to pad the record 123 on the right with three blanks. If BZ editing is in effect, those three blanks are interpreted as zeros, and the record is equal to 123000.

Scale-Factor Editing (P)

The P edit descriptor specifies a scale factor, which moves the location of the decimal point in real values and the two real parts of complex values. It takes the following form:

*k*P

The *k* is a signed (sign is optional if positive), integer literal constant specifying the number of positions, to the left or right, that the decimal point is to move (the scale factor). The range of *k* is -128 to 127.

At the beginning of a formatted I/O statement, the value of the scale factor is zero. If a scale editing descriptor is specified, the scale factor is set to the new value, which affects all subsequent real edit descriptors until another scale editing descriptor occurs.

To reinstate a scale factor of zero, you must explicitly specify 0P.

Format reversion does not affect the scale factor. (For more information on format reversion, see Interaction Between Format Specifications and I/O Lists.)

Rules for Input Processing

On input, a positive scale factor moves the decimal point to the left, and a negative scale factor moves the decimal point to the right. (On output, the effect is the reverse.)

On input, when an input field using an F, E, D, EN, ES, or G real edit descriptor contains an explicit exponent, the scale factor has no effect. Otherwise, the internal value of the corresponding I/O list item is equal to the external field data multiplied by 10^{-k}. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left. A -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right.

The following shows input using the P edit descriptor (the symbol ^ represents a nonprinting blank character):

```
Format     Input          Value

3PE10.5    ^^^37.614^           .037614

3PE10.5    ^^37.614E2      3761.4

-3PE10.5   ^^^^37.614     37614.0
```

The scale factor must precede the first real edit descriptor associated with it, but it need not immediately precede the descriptor. For example, the following all have the same effect:

```
(3P, I6, F6.3, E8.1)

(I6, 3P, F6.3, E8.1)

(I6, 3PF6.3, E8.1)
```

Note that if the scale factor immediately precedes the associated real edit descriptor, the comma separator is optional.

Rules for Output Processing

On output, a positive scale factor moves the decimal point to the right, and a negative scale factor moves the decimal point to the left. (On input, the effect is the reverse.)

On output, the effect of the scale factor depends on which kind of real editing is associated with it, as follows:

- For F editing, the external value equals the internal value of the I/O list item multiplied by 10^{k}. This changes the magnitude of the data.
- For E and D editing, the external decimal field of the I/O list item is multiplied by 10^{k}, and k is subtracted from the exponent. This changes the form of the data.

A positive scale factor decreases the exponent; a negative scale factor increases the exponent.

For a positive scale factor, *k* must be less than *d* + 2 or an output conversion error occurs.

- For G editing, the scale factor has no effect if the magnitude of the data to be output is within the effective range of the descriptor (the G descriptor supplies its own scaling).

  If the magnitude of the data field is outside G descriptor range, E editing is used, and the scale factor has the same effect as E output editing.
- For EN and ES editing, the scale factor has no effect.

The following shows output using the P edit descriptor (the symbol ^ represents a nonprinting blank character):

Format	Value	Output
1PE12.3	-270.139	^^-2.701E+02
1P,E12.2	-270.139	^^^-2.70E+02
-1PE12.2	-270.139	^^^-0.03E+04

Examples

The following shows a FORMAT statement containing a scale factor:

```
     DIMENSION A(6)
     DO 10 I=1,6
10   A(I) = 25.

     WRITE (6, 100) A
100  FORMAT(' ', F8.2, 2PF8.2, F8.2)
```

The preceding statements produce the following results:

```
    25.00  2500.00  2500.00
  2500.00  2500.00  2500.00
```

The following code uses scale-factor editing when reading:

```
       READ (*, 100) a, b, c, d
 100   FORMAT (F10.6, 1P, F10.6, F10.6, -2P, F10.6)
       WRITE (*, 200) a, b, c, d
 200   FORMAT (4F11.3)
```

If the following data is entered:

```
 12340000  12340000  12340000  12340000
    12.34     12.34     12.34     12.34
  12.34e0   12.34e0   12.34e0   12.34e0
  12.34e3   12.34e3   12.34e3   12.34e3
```

The program's output is:

```
    12.340      1.234      1.234   1234.000
    12.340      1.234      1.234   1234.000
    12.340     12.340     12.340     12.340
 12340.000  12340.000  12340.000  12340.000
```

The next code shows scale-factor editing when writing:

```
       a = 12.34
       WRITE (*, 100) a, a, a, a, a, a
 100   FORMAT (1X, F9.4, E11.4E2, 1P, F9.4, E11.4E2, &
      &        -2P, F9.4, E11.4E2)
```

This program's output is:

```
 12.3400 0.1234E+02 123.4000 1.2340E+01   0.1234 0.0012E+04
```

See Also

- Control Edit Descriptors
- Forms for Control Edit Descriptors

Slash Editing (/)

The slash edit descriptor terminates data transfer for the current record and starts data transfer for a new record. It takes the following form:

[r]/

The *r* is a repeat specification. It must be a positive default integer literal constant; no kind parameter can be specified.

The range of *r* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture and IA-64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. If *r* is omitted, it is assumed to be 1.

Multiple slashes cause the system to skip input records or to output blank records, as follows:

- When *n* consecutive slashes appear between two edit descriptors, *n* - 1 records are skipped on input, or *n* - 1 blank records are output. The first slash terminates the current record. The second slash terminates the first skipped or blank record, and so on.
- When *n* consecutive slashes appear at the beginning or end of a format specification, *n* records are skipped or *n* blank records are output, because the opening and closing parentheses of the format specification are themselves a record initiator and terminator, respectively. For example, suppose the following statements are specified:

```
     WRITE (6,99)
99   FORMAT ('1',T51,'HEADING LINE'//T51,'SUBHEADING LINE'//)
```

The following lines are written:

```
                  Column 50, top of page         |         HEADING LINE
(blank line)          SUBHEADING LINE
(blank line)
(blank line)
```

Note that the first character of the record printed was reserved as a control character (see Printing of Formatted Records).

Examples

```
!     The following statements write spreadsheet column and row labels:
      WRITE (*, 100)
100   FORMAT ('   A     B     C     D     E'                              &
     &        /,' 1',/,' 2',/,' 3',/,' 4',/,' 5')
```

This example generates the following output:

```
  A     B     C     D     E
1
2
3
4
5
```

See Also

- Control Edit Descriptors
- Forms for Control Edit Descriptors

Colon Editing (:)

The colon edit descriptor terminates format control if no more items are in the I/O list.

Examples

Suppose the following statements are specified:

```
      PRINT 1,3
      PRINT 2,13
1     FORMAT (' I=',I2,' J=',I2)
2     FORMAT (' K=',I2,:,' L=',I2)
```

The following lines are written (the symbol ^ represents a nonprinting blank character):

```
I=^3^J=
K=13
```

```
 !     The following example writes a= 3.20 b= .99
       REAL a, b, c, d
       DATA a /3.2/, b /.9871515/
       WRITE (*, 100) a, b
 100   FORMAT (' a=', F5.2, :, ' b=', F5.2, :, &
      &        ' c=', F5.2, :, ' d=', F5.2)
       END
```

See Also

- Control Edit Descriptors
- Forms for Control Edit Descriptors

Dollar-Sign ($) and Backslash (\ ) Editing

The dollar sign and backslash edit descriptors modify the output of carriage control specified by the first character of the record. They only affect carriage control for formatted files, and have no effect on input.

If the first character of the record is a blank or a plus sign (+), the dollar sign and backslash descriptors suppress carriage return (after printing the record).

For terminal device I/O, when this trailing carriage return is suppressed, a response follows output on the same line. For example, suppose the following statements are specified:

```
       TYPE 100
100    FORMAT (' ENTER RADIUS VALUE ',$)
       ACCEPT 200, RADIUS
200    FORMAT (F6.2)
```

The following prompt is displayed:

```
  ENTER RADIUS VALUE
```

Any response (for example, "12.") is then displayed on the same line:

```
  ENTER RADIUS VALUE    12.
```

If the first character of the record is 0, 1, or ASCII NUL, the dollar sign and backslash descriptors have no effect.

Consider the following:

```
     CHARACTER(20) MYNAME
     WRITE (*,9000)
9000 FORMAT ('0Please type your name:',\)
     READ  (*,9001) MYNAME
9001 FORMAT (A20)
     WRITE (*,9002) ' ',MYNAME
9002 FORMAT (1X,A20)
```

This example advances two lines, prompts for input, awaits input on the same line as the prompt, and prints the input.

See Also

- Control Edit Descriptors
- Forms for Control Edit Descriptors

Character Count Editing (Q)

The character count edit descriptor returns the remaining number of characters in the current input record.

The corresponding I/O list item must be of type integer or logical. For example, suppose the following statements are specified:

```
      READ (4,1000) XRAY, KK, NCHRS, (ICHR(I), I=1,NCHRS)

1000  FORMAT (E15.7,I4,Q,(80A1))
```

Two fields are read into variables XRAY and KK. The number of characters remaining in the record is stored in NCHRS, and exactly that many characters are read into the array ICHR. (This instruction can fail if the record is longer than 80 characters.)

If you place the character count descriptor first in a format specification, you can determine the length of an input record.

On output, the character count edit descriptor causes the corresponding I/O list item to be skipped.

Examples

Consider the following:

```
       CHARACTER ICHAR(80)

       READ (4, 1000) XRAY, K, NCHAR, (ICHAR(I), I= 1, NCHAR)

 1000  FORMAT (E15.7, I4, Q, 80A1)
```

The preceding input statement reads the variables XRAY and K. The number of characters remaining in the record is NCHAR, specified by the Q edit descriptor. The array ICHAR is then filled by reading exactly the number of characters left in the record. (Note that this instruction will fail if NCHAR is greater than 80, the length of the array ICHAR.) By placing Q in the format specification, you can determine the actual length of an input record.

Note that the length returned by Q is the number of characters left in the record, not the number of reals or integers or other data types. The length returned by Q can be used immediately after it is read and can be used later in the same format statement or in a variable format expression. (See Variable Format Expressions.)

Assume the file Q.DAT contains:

```
1234.567Hello, Q Edit
```

The following program reads in the number REAL1, determines the characters left in the record, and reads those into STR:

```
       CHARACTER STR(80)
       INTEGER LENGTH
       REAL REAL1
       OPEN (UNIT = 10, FILE = 'Q.DAT')
 100   FORMAT (F8.3, Q, 80A1)
       READ (10, 100) REAL1, LENGTH, (STR(I), I=1, LENGTH)
       WRITE(*,'(F8.3,2X,I2,2X,<LENGTH>A1)') REAL1, LENGTH, (STR(I), &
      & I= 1, LENGTH)
       END
```

The output on the screen is:

```
1234.567  13  Hello, Q Edit
```

A READ statement that contains only a Q edit descriptor advances the file to the next record. For example, consider that Q.DAT contains the following data:

```
abcdefg
abcd
```

Consider it is then READ with the following statements:

```
       OPEN (10, FILE = "Q.DAT")
       READ(10, 100) LENGTH
 100   FORMAT(Q)
       WRITE(*,'(I2)') LENGTH
       READ(10, 100) LENGTH
       WRITE(*,'(I2)') LENGTH
       END
```

The output to the screen would be:

```
7
4
```

See Also

- Control Edit Descriptors
- Forms for Control Edit Descriptors

Character String Edit Descriptors

Character string edit descriptors control the output of character strings. The character string edit descriptors are the character constant and H edit descriptor.

Although no string edit descriptor can be preceded by a repeat specification, a parenthesized group of string edit descriptors can be preceded by a repeat specification.

See Also

- I/O Formatting
- Character Constant Editing
- H Editing
- Nested and Group Repeat Specifications

Character Constant Editing

The character constant edit descriptor causes a character string to be output to an external record. It takes one of the following forms:

'*string*'

"*string*"

The *string* is a character literal constant; no kind parameter can be specified. Its length is the number of characters between the delimiters; two consecutive delimiters are counted as one character.

To include an apostrophe in a character constant that is enclosed by apostrophes, place two consecutive apostrophes ('') in the format specification; for example:

```
50   FORMAT ('TODAY''S^DATE^IS:^',I2,'/',I2,'/',I2)
```

Note that the symbol ^ represents a nonprinting blank character.

Similarly, to include a quotation mark in a character constant that is enclosed by quotation marks, place two consecutive quotation marks ("") in the format specification.

On input, the character constant edit descriptor transfers length of string characters to the edit descriptor.

Examples

Consider the following '(3I5)' format in the WRITE statement:

```
WRITE (10, '(3I5)') I1, I2, I3
```

This is equivalent to:

```
      WRITE (10, 100) I1, I2, I3
100   FORMAT( 3I5)
```

The following shows another example:

```
!     These WRITE statements both output ABC'DEF
!     (The leading blank is a carriage-control character).
      WRITE (*, 970)
970   FORMAT (' ABC''DEF')
      WRITE (*, '('' ABC''''DEF'')')
!     The following WRITE also outputs ABC'DEF. No carriage-
!     control character is necessary for list-directed I/O.
      WRITE (*,*) 'ABC''DEF'
```

Alternatively, if the delimiter is quotation marks, the apostrophe in the character constant `ABC'DEF` requires no special treatment:

```
WRITE (*,*) "ABC'DEF"
```

See Also

- Character String Edit Descriptors
- Character constants
- Format Specifications

H Editing

The H edit descriptor transfers data between the external record and the H edit descriptor itself. The H edit descriptor is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel® Fortran fully supports features deleted in Fortran 95.

An H edit descriptor has the form of a Hollerith constant, as follows:

*n*H*string*

n	Is an unsigned, positive default integer literal constant (with no kind parameter) indicating the number of characters in *string* (including blanks and tabs). The range of *n* is 1 through 2147483647 (2**31-1) on Intel® 64 architecture and IA-64 architecture; 1 through 32767 (2**15-1) on IA-32 architecture. Actual useful ranges may be constrained by record sizes (RECL) and the file system.
string	Is a string of printable ASCII characters.

On input, the H edit descriptor transfers *n* characters from the external field to the edit descriptor. The first character appears immediately after the letter H. Any characters in the edit descriptor before input are replaced by the input characters.

On output, the H edit descriptor causes *n* characters following the letter H to be output to an external record.

Examples

```
!     These WRITE statements both print "Don't misspell 'Hollerith'"
!     (The leading blanks are carriage-control characters).
!     Hollerith formatting does not require you to embed additional
!     single quotation marks as shown in the second example.
!
      WRITE (*, 960)
960   FORMAT (27H Don't misspell 'Hollerith')
      WRITE (*, 961)
961   FORMAT (' Don''t misspell ''Hollerith''')
```

See Also

- Character String Edit Descriptors
- Obsolescent and Deleted Language Features
- Format Specifications

Nested and Group Repeat Specifications

Format specifications can include nested format specifications enclosed in parentheses; for example:

```
15   FORMAT (E7.2,I8,I2,(A5,I6))

35   FORMAT (A6,(L8(3I2)),A)
```

A group repeat specification can precede a nested group of edit descriptors. For example, the following statements are equivalent, and the second statement shows a group repeat specification:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7,I5,I5)

50   FORMAT (2I8,3(F8.3,E15.7),2I5)
```

If a nested group does not show a repeat count, a default count of 1 is assumed.

Normally, the string edit descriptors and control edit descriptors cannot be repeated (except for slash), but any of these descriptors can be enclosed in parentheses and preceded by a group repeat specification. For example, the following statements are valid:

```
76   FORMAT ('MONTHLY',3('TOTAL'))

100  FORMAT (I8,4(T7),A4)
```

See Also

- I/O Formatting
- string edit descriptors
- control edit descriptors
- Forms for Data Edit Descriptors
- Interaction Between Format Specifications and I/O Lists

Variable Format Expressions

A variable format expression is a numeric expression enclosed in angle brackets (< >) that can be used in a FORMAT statement or in character format specifications.

The numeric expression can be any valid Fortran expression, including function calls and references to dummy arguments.

If the expression is not of type integer, it is converted to integer type before being used.

If the value of a variable format expression does not obey the restrictions on magnitude applying to its use in the format, an error occurs.

Variable format expressions cannot be used with the H edit descriptor, and they are not allowed in character format specifications.

Variable format expressions are evaluated each time they are encountered in the scan of the format. If the value of the variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Examples

Consider the following statement:

```
  FORMAT (I<J+1>)
```

When the format is scanned, the preceding statement performs an I (integer) data transfer with a field width of J+1. The expression is reevaluated each time it is encountered in the normal format scan.

Consider the following statements:

```
      DIMENSION A(5)
      DATA A/1.,2.,3.,4.,5./
      DO 10 I=1,10
      WRITE (6,100) I
100   FORMAT (I<MAX(I,5)>)
10    CONTINUE
      DO 20 I=1,5
      WRITE (6,101) (A(I), J=1,I)
101   FORMAT (<I>F10.<I-1>)
20    CONTINUE
      END
```

On execution, these statements produce the following output:

```
     1
     2
     3
     4
     5
      6
       7
        8
         9
         10
         1.
        2.0         2.0
       3.00        3.00      3.00
      4.000       4.000     4.000      4.000
     5.0000     5.0000     5.0000     5.0000     5.0000
```

The following shows another example:

```
        WRITE(6,20) INT1
 20     FORMAT(I<MAX(20,5)>)
        WRITE(6,FMT=30) REAL2(10), REAL3
 30     FORMAT(<J+K>X, <2*M>F8.3)
```

The value of the expression is reevaluated each time an input/output item is processed during the execution of the READ, WRITE, or PRINT statement. For example:

```
        INTEGER width, value
        width=2
        READ (*,10) width, value
 10     FORMAT(I1, I <width>)
        PRINT *, value
        END
```

When given input 3123, the program will print 123 and not 12.

See Also

- I/O Formatting
- Interaction Between Format Specifications and I/O Lists

Printing of Formatted Records

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect or the file is being processed by the `fortpr` format utility, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but used to control vertical spacing.

The following table lists the valid control characters for printing:

Table 622: Control Characters for Printing

Character	Meaning	Effect
+	Overprinting	Outputs the record (at the current position in the current line) and a carriage return.
-	One line feed	Outputs the record (at the beginning of the following line) and a carriage return.
0	Two line feeds	Outputs the record (after skipping a line) and a carriage return.
1	Next page	Outputs the record (at the beginning of a new page) and a carriage return.
$	Prompting	Outputs the record (at the beginning of the following line), but no carriage return.
ASCII NUL[1]	Overprinting with no advance	Outputs the record (at the current position in the current line), but no carriage return.

Specify as CHAR(0).

Any other character is interpreted as a blank and is deleted from the print line. If you do not specify a control character for printing, the first character of the record is not printed.

Interaction Between Format Specifications and I/O Lists

Format control begins with the execution of a formatted I/O statement. Each action of format control depends on information provided jointly by the next item in the I/O list (if one exists) and the next edit descriptor in the format specification.

Both the I/O list and the format specification are interpreted from left to right, unless repeat specifications or implied-DO lists appear.

If an I/O list specifies at least one list item, at least one data edit descriptor (I, B, O, Z, F, E, EN, ES, D, G, L, or A) or the Q edit descriptor must appear in the format specification; otherwise, an error occurs.

Each data edit descriptor (or Q edit descriptor) corresponds to one item in the I/O list, except that an I/O list item of type complex requires the interpretation of two F, E, EN, ES, D, or G edit descriptors. No I/O list item corresponds to a control edit descriptor (X, P, T, TL, TR, SP, SS, S, BN, BZ, $, or :), or a character string edit descriptor (H and character constants). For character string edit descriptors, data transfer occurs directly between the external record and the format specification.

When format control encounters a data edit descriptor in a format specification, it determines whether there is a corresponding I/O list item specified. If there is such an item, it is transferred under control of the edit descriptor, and then format control proceeds. If there is no corresponding I/O list item, format control terminates.

If there are no other I/O list items to be processed, format control also terminates when the following occurs:

- A colon edit descriptor is encountered.
- The end of the format specification is reached.

If additional I/O list items remain, part or all of the format specification is reused in format reversion.

In format reversion, the current record is terminated and a new one is initiated. Format control then reverts to one of the following (in order) and continues from that point:

1. The group repeat specification whose opening parenthesis matches the next-to-last closing parenthesis of the format specification
2. The initial opening parenthesis of the format specification

Format reversion has no effect on the scale factor (P), the sign control edit descriptors (S, SP, or SS), or the blank interpretation edit descriptors (BN or BZ).

Examples

The data in file FOR002.DAT is to be processed 2 records at a time. Each record starts with a number to be put into an element of a vector B, followed by 5 numbers to be put in a row in matrix A.

FOR002.DAT contains the following data:

```
001 0101 0102 0103 0104 0105
002 0201 0202 0203 0204 0205
003 0301 0302 0303 0304 0305
004 0401 0402 0403 0404 0405
005 0501 0502 0503 0504 0505
006 0601 0602 0603 0604 0605
007 0701 0702 0703 0704 0705
008 0801 0802 0803 0804 0805
009 0901 0902 0903 0904 0905
010 1001 1002 1003 1004 1005
```

The following example shows how several different format specifications interact with I/O lists to process data in file FOR002.DAT:

Figure 34: Interaction Between Format Specifications and I/O Lists

```
        INTEGER I, J, A(2,5), B(2)
        OPEN (unit=2, access='sequential', file='FOR002.DAT')
        READ (2,100) (B(I), (A(I,J), J=1,5),I=1,2)
1
  100   FORMAT (2 (I3, X, 5(I4,X), /) )
2
        WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
```

3

```
 999   FORMAT (' B is ', 2(I3, X), ';  A is', /
     1       (' ', 5 (I4, X)) )
       READ (2,200) (B(I), (A(I,J), J=1,5),I=1,2)
```

4

```
 200   FORMAT (2 (I3, X, 5(I4,X), :/) )
      WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
```

5

```
       READ (2,300) (B(I), (A(I,J), J=1,5),I=1,2)
```

6

```
 300   FORMAT ( (I3, X, 5(I4,X)) )
       WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
```

7

```
       READ (2,400) (B(I), (A(I,J), J=1,5),I=1,2)
```

8

```
 400   FORMAT ( I3, X, 5(I4,X) )
       WRITE (6,999) B, ((A(I,J),J=1,5),I=1,2)
```

9

```
      END
```

1 This statement reads B(1); then A(1,1) through A(1,5); then B(2) and A(2,1) through A(2,5).

The first record read (starting with 001) starts the processing of the I/O list.

2 There are two records, each in the format I3, X, 5(I4, X). The slash (/) forces the reading of the second record after A(1,5) is processed. It also forces the reading of the third record after A(2,5) is processed; no data is taken from that record.

3 This statement produces the following output:

```
B is   1   2 ;  A is 101  102  103  104  105 201  202  203  204  205
```

4 This statement reads the record starting with 004. The slash (/) forces the reading of the next record after A(1,5) is processed. The colon (:) stops the reading after A(2,5) is processed, but before the slash (/) forces another read.

5 This statement produces the following output:

```
B is   4   5 ;  A is 401  402  403  404  405 501  502  503  504  505
```

6 This statement reads the record starting with 006. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I3.

7 This statement produces the following output:

```
B is   6   7 ;  A is 601  602  603  604  605 701  702  703  704  705
```

8 This statement reads the record starting with 008. After A(1,5) is processed, format reversion causes the next record to be read and starts format processing at the left parenthesis before the I4.

9 This statement produces the following output:

```
B is   8  90 ;  A is 801  802  803  804  805 9010 9020 9030 9040 100
```

The record 009 0901 0902 0903 0904 0905 is processed with I4 as "009 " for B(2), which is 90. X skips the next "0". Then "901 " is processed for A(2,1), which is 9010, "902 " for A(2,2), "903 " for A(2,3), and "904 " for A(2,4). The repeat specification of 5 is now exhausted and the format ends. Format reversion causes another record to be read and starts format processing at the left parenthesis before the I4, so "010 " is read for A(2,5), which is 100.

See Also

- I/O Formatting
- Data edit descriptors
- Control edit descriptors
- Q edit descriptor
- Character string edit descriptors
- Scale Factor Editing (P)

File Operation I/O Statements

53

The following are file connection, inquiry, and positioning I/O statements:

- BACKSPACE

  Positions a sequential file at the beginning of the preceding record.
- CLOSE

  Terminates the connection between a logical unit and a file or device.
- DELETE

  Deletes a record from a relative file.
- ENDFILE

  For sequential files, writes an end-of-file record to the file and positions the file after this record. For direct access files, truncates the file after the current record.
- FLUSH

  Causes data written to a file to become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement.
- INQUIRE

  Requests information on the status of specified properties of a file or logical unit.
- OPEN

  Connects a Fortran logical unit to a file or device; declares attributes for read and write operations.
- REWIND

  Positions a sequential file at the beginning of the file.
- WAIT

  Performs a wait operation for a specified pending asynchronous data transfer operation.

The following table summarizes I/O statement specifiers:

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
ACCESS=*access*	'SEQUENTIAL', 'DIRECT', 'STREAM', or 'APPEND'	Specifies the method of file access.	INQUIRE, OPEN

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
ACTION=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Specifies file I/O mode.	INQUIRE, OPEN
ADVANCE=*c-expr*	'NO' or 'YES' (default is 'YES')	Specifies formatted sequential data input as advancing, or non-advancing.	READ
ASSOCIATEVARIABLE=*var*	Integer variable	Specifies a variable to be updated to reflect the record number of the next sequential record in the file.	OPEN
ASYNCHRONOUS=*asynch*	'YES' or 'NO' (default is 'NO')	Specifies whether or not the I/O is done asynchronously	INQUIRE, OPEN
BINARY=*bin*	'NO' or 'YES'	Returns whether file format is binary.	INQUIRE
BLANK=*blank_control*	'NULL' or 'ZERO' (default is 'NULL')	Specifies whether blanks are ignored in numeric fields or interpreted as zeros.	INQUIRE, OPEN
BLOCKSIZE=*blocksize*	Positive integer variable or expression	Specifies or returns the internal buffer size used in I/O.	INQUIRE, OPEN
BUFFERCOUNT=*bc*	Numeric expression	Specifies the number of buffers to be associated with the unit for multibuffered I/O.	OPEN

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
BUFFERED=*bf*	'YES' or 'NO' (default is 'NO')	Specifies run-time library behavior following WRITE operations.	INQUIRE, OPEN
CARRIAGECONTROL=*control*	'FORTRAN', 'LIST', or 'NONE'	Specifies carriage control processing.	INQUIRE, OPEN
CONVERT=*form*	'LITTLE_ENDIAN', 'BIG_ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG', or 'NATIVE' (default is 'NATIVE')	Specifies a numeric format for unformatted data.	INQUIRE, OPEN
DEFAULTFILE=*var*	Character expression	Specifies a default file pathname string.	INQUIRE, OPEN
DELIM=*delimiter*	'APOSTROPHE', 'QUOTE' or 'NONE' (default is 'NONE')	Specifies the delimiting character for list-directed or namelist data.	INQUIRE, OPEN
DIRECT=*dir*	'NO' or 'YES'	Returns whether file is connected for direct access.	INQUIRE
DISPOSE=*dis* (or DISP=*dis*)	'KEEP', 'SAVE', 'DELETE', 'PRINT', 'PRINT/DELETE', 'SUBMIT', or 'SUBMIT/DELETE' (default is 'DELETE' for scratch files; 'KEEP' for all other files)	Specifies the status of a file after the unit is closed.	OPEN, CLOSE
formatlist	Character variable or expression	Lists edit descriptors. Used in FORMAT statements and format specifiers (the	FORMAT, PRINT, READ, WRITE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
		FMT=*formatspec* option) to describe the format of data.	
END=*endlabel*	Integer between 1 and 99999	When an end of file is encountered, transfers control to the statement whose label is specified.	READ
EOR=*eorlabel*	Integer between 1 and 99999	When an end of record is encountered, transfers to the statement whose label is specified.	READ
ERR=*errlabel*	Integer between 1 and 99999	Specifies the label of an executable statement where execution is transferred after an I/O error.	All except PRINT
EXIST=*ex*	.TRUE. or .FALSE.	Returns whether a file exists and can be opened.	INQUIRE
FILE=*file* (or NAME=*name*)	Character variable or expression. Length and format of the name are determined by the operating system	Specifies the name of a file	INQUIRE, OPEN
[FMT=]*formatspec*	Character variable or expression	Specifies an *editlist* to use to format data.	PRINT, READ, WRITE
FORM=*form*	'FORMATTED', 'UNFORMATTED', or 'BINARY'	Specifies a file's format.	INQUIRE, OPEN

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
FORMATTED=*fmt*	'NO' or 'YES'	Returns whether a file is connected for formatted data transfer.	INQUIRE
IOFOCUS=*iof*	.TRUE. or .FALSE. (default is .TRUE. unless unit '*' is specified)	Specifies whether a unit is the active window in a QuickWin application.	INQUIRE, OPEN
iolist	List of variables of any type, character expression, or NAMELIST	Specifies items to be input or output.	PRINT, READ, WRITE
IOSTAT=*iostat*	Integer variable	Specifies a variable whose value indicates whether an I/O error has occurred.	All except PRINT
MAXREC=*var*	Numeric expression	Specifies the maximum number of records that can be transferred to or from a direct access file.	OPEN
MODE=*permission*	'READ', 'WRITE' or 'READWRITE' (default is 'READWRITE')	Same as ACTION.	INQUIRE, OPEN
NAMED=*var*	.TRUE. or .FALSE.	Returns whether a file is named.	INQUIRE
NEXTREC=*nr*	Integer variable	Returns where the next record can be read or written in a file.	INQUIRE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
[NML=]*nmlspec*	Namelist name	Specifies a *namelist* group to be input or output.	PRINT, READ, WRITE
NUMBER=*num*	Integer variable	Returns the number of the unit connected to a file.	INQUIRE
OPENED=*od*	.TRUE. or .FALSE.	Returns whether a file is connected.	INQUIRE
ORGANIZATION=*org*	'SEQUENTIAL' or 'RELATIVE' (default is 'SEQUENTIAL')	Specifies the internal organization of a file.	INQUIRE, OPEN
PAD=*pad_switch*	'YES' or 'NO' (default is 'YES')	Specifies whether an input record is padded with blanks when the input list or format requires more data than the record holds, or whether the input record is required to contain the data indicated.	INQUIRE, OPEN
POS=*pos*	Positive integer	Specifies the file storage unit position in a stream file.	INQUIRE, READ, WRITE
POSITION=*file_pos*	'ASIS', 'REWIND' or 'APPEND' (default is 'ASIS')	Specifies position in a file.	INQUIRE, OPEN
READ=*rd*	'NO' or 'YES'	Returns whether a file can be read.	INQUIRE

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
READONLY		Specifies that only READ statements can refer to this connection.	OPEN
READWRITE=*rdwr*	'NO' or 'YES'	Returns whether a file can be both read and written to.	INQUIRE
REC=*rec*	Positive integer variable or expression	Specifies the first (or only) record of a file to be read from, or written to.	READ, WRITE
RECL=*length* (or RECORDSIZE=*length*)	Positive integer variable or expression	Specifies the record length in direct access files, or the maximum record length in sequential files.	INQUIRE, OPEN
RECORDTYPE=*typ*	'FIXED', 'VARIABLE', 'SEGMENTED', 'STREAM', 'STREAM_LF', or 'STREAM_CR'	Specifies the type of records in a file.	INQUIRE, OPEN
SEQUENTIAL=*seq*	'NO' or 'YES'	Returns whether file is connected for sequential access.	INQUIRE
SHARE=*share*	'COMPAT', 'DENYNONE', 'DENYWR', 'DENYRD', or 'DENYRW' (default is 'DENYNONE')	Controls how other processes can simultaneously access a file on networked systems.	INQUIRE, OPEN

I/O Specifiers			
Specifier	**Values**	**Description**	**Used with:**
SHARED		Specifies that a file is connected for shared access by more than one program executing simultaneously.	OPEN
SIZE=*size*	Integer variable	Returns the number of characters read in a nonadvancing READ before an end-of-record condition occurred.	READ
STATUS=*status* (or TYPE=*status*)	'OLD', 'NEW', 'UNKNOWN' or 'SCRATCH' (default is 'UNKNOWN')	Specifies the status of a file on opening and/or closing.	CLOSE, OPEN
TITLE=*name*	Character expression	Specifies the name of a child window in a QuickWin application.	OPEN
UNFORMATTED=*unf*	'NO' or 'YES'	Returns whether a file is connected for unformatted data transfer.	INQUIRE
[UNIT=]*unitspec*	Integer variable or expression	Specifies the unit to which a file is connected.	All except PRINT
USEROPEN=*fname*	Name of a user-written function	Specifies an external function that controls the opening of a file.	OPEN
WRITE=*rd*	'NO' or 'YES'	Returns whether a file can be written to.	INQUIRE

BACKSPACE Statement Overview

The BACKSPACE statement positions a file at the beginning of the preceding record, making it available for subsequent I/O processing.

CLOSE Statement Overview

The CLOSE statement disconnects a file from a unit.

DELETE Statement Overview

The DELETE statement deletes a record from a relative file.

ENDFILE Statement Overview

The ENDFILE statement writes an end-of-file record to a sequential file and positions the file after this record (the terminal point), or it causes a direct access file to be truncated after the current record.

FLUSH Statement Overview

The FLUSH statement makes data written to a file become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement. For more information, see FLUSH.

INQUIRE Statement Overview

The INQUIRE statement returns information on the status of specified properties of a file or logical unit. For more information, see INQUIRE.

The following are inquiry specifiers:

ACCESS	DELIM	NAMED	READ
ACTION	DIRECT	NEXTREC	READWRITE
ASYNCHRONOUS	EXIST	NUMBER	RECL
BINARY	FORM	OPENED	RECORDTYPE

BLANK	FORMATTED	ORGANIZATION	SEQUENTIAL
BLOCKSIZE	ID	PAD	SHARE
BUFFERED	IOFOCUS	PENDING	UNFORMATTED
CARRIAGECONTROL	MODE	POS	WRITE
CONVERT	NAME	POSITION	

INQUIRE: ACCESS Specifier

The ACCESS specifier asks how a file is connected. It takes the following form:

ACCESS = *acc*

acc Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is connected for sequential access
'STREAM'	If the file is connected for stream access
'DIRECT'	If the file is connected for direct access
'UNDEFINED'	If the file is not connected

INQUIRE: ACTION Specifier

The ACTION specifier asks which I/O operations are allowed for a file. It takes the following form:

ACTION = *act*

act Is a scalar default character variable that is assigned one of the following values:

'READ'	If the file is connected for input only
'WRITE'	If the file is connected for output only

'READWRITE'	If the file is connected for both input and output
'UNDEFINED'	If the file is not connected

INQUIRE: ASYNCHRONOUS Specifier

The ASYNCHRONOUS specifier asks whether asynchronous I/O is in effect. It takes the following form:

ASYNCHRONOUS = *asyn*

asyn Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit is connected and asynchronous input/output is not in effect.
'YES'	If the file or unit is connected and asynchronous input/output is in effect.
'UNKNOWN'	If the file or unit is not connected.

INQUIRE: BINARY Specifier (W*32, W*64)

The BINARY specifier asks whether a file is connected to a binary file. It takes the following form:

BINARY = *bin*

bin Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected to a binary file
'NO'	If the file is connected to a nonbinary file
'UNKNOWN'	If the file is not connected

INQUIRE: BLANK Specifier

The BLANK specifier asks what type of blank control is in effect for a file. It takes the following form:

BLANK = *blnk*

blnk Is a scalar default character variable that is assigned one of the following values:

'NULL'	If null blank control is in effect for the file
'ZERO'	If zero blank control is in effect for the file
'UNDEFINED'	If the file is not connected, or it is not connected for formatted data transfer

INQUIRE: BLOCKSIZE Specifier

The BLOCKSIZE specifier asks about the I/O buffer size. It takes the following form:

BLOCKSIZE = *bks*

bks Is a scalar integer variable.
The *bks* is assigned the current size of the I/O buffer. If the unit or file is not connected, the value assigned is zero.

INQUIRE: BUFFERED Specifier

The BUFFERED specifier asks whether run-time buffering is in effect. It takes the following form:

BUFFERED = *bf*

bf Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit is connected and buffering is not in effect.
'YES'	If the file or unit is connected and buffering is in effect.

'UNKNOWN'	If the file or unit is not connected.

INQUIRE: CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier asks what type of carriage control is in effect for a file. It takes the following form:

CARRIAGECONTROL = *cc*

cc Is a scalar default character variable that is assigned one of the following values:

'FORTRAN'	If the file is connected with Fortran carriage control in effect
'LIST'	If the file is connected with implied carriage control in effect
'NONE'	If the file is connected with no carriage control in effect
'UNKNOWN'	If the file is not connected

INQUIRE: CONVERT Specifier

The CONVERT specifier asks what type of data conversion is in effect for a file. It takes the following form:

CONVERT = *fm*

fm Is a scalar default character variable that is assigned one of the following values:

'LITTLE_ENDIAN'	If the file is connected with little endian integer and IEEE* floating-point data conversion in effect
'BIG_ENDIAN'	If the file is connected with big endian integer and IEEE floating-point data conversion in effect

'CRAY'	If the file is connected with big endian integer and CRAY* floating-point data conversion in effect
'FDX'	If the file is connected with little endian integer and VAX* processor F_floating, D_floating, and IEEE X_floating data conversion in effect
'FGX'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and IEEE X_floating data conversion in effect
'IBM'	If the file is connected with big endian integer and IBM* System\370 floating-point data conversion in effect
'VAXD'	If the file is connected with little endian integer and VAX processor F_floating, D_floating, and H_floating in effect
'VAXG'	If the file is connected with little endian integer and VAX processor F_floating, G_floating, and H_floating in effect
'NATIVE'	If the file is connected with no data conversion in effect
'UNKNOWN'	If the file or unit is not connected for unformatted data transfer

INQUIRE: DELIM Specifier

The DELIM specifier asks how character constants are delimited in list-directed and namelist output. It takes the following form:

DELIM = *del*

del Is a scalar default character variable that is assigned one of the following values:

'APOSTROPHE'	If apostrophes are used to delimit character constants in list-directed and namelist output
'QUOTE'	If quotation marks are used to delimit character constants in list-directed and namelist output
'NONE'	If no delimiters are used
'UNDEFINED'	If the file is not connected, or is not connected for formatted data transfer

INQUIRE: DIRECT Specifier

The DIRECT specifier asks whether a file is connected for direct access. It takes the following form:

DIRECT = *dir*

dir Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for direct access
'NO'	If the file is not connected for direct access
'UNKNOWN'	If the file is not connected

INQUIRE: EXIST Specifier

The EXIST specifier asks whether a file exists and can be opened. It takes the following form:

EXIST = *ex*

ex Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file exists and can be opened, or if the specified unit exists
.FALSE.	If the specified file or unit does not exist or if the file exists but cannot be opened

The unit exists if it is a number in the range allowed by the processor.

INQUIRE: FORM Specifier

The FORM specifier asks whether a file is connected for formatted, unformatted, or binary (W*32, W*64) data transfer. It takes the following form:

FORM = *fm*

fm Is a scalar default character variable that is assigned one of the following values:

'FORMATTED'	If the file is connected for formatted data transfer
'UNFORMATTED'	If the file is connected for unformatted data transfer
'BINARY'	If the file is connected for binary data transfer
'UNDEFINED'	If the file is not connected

INQUIRE: FORMATTED Specifier

The FORMATTED specifier asks whether a file is connected for formatted data transfer. It takes the following form:

FORMATTED = *fmt*

fmt Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for formatted data transfer

'NO'	If the file is not connected for formatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for formatted data transfer

INQUIRE: ID Specifier

The ID specifier identifies a pending asynchronous data transfer. It takes the following form:

ID=*iexp*

iexp	Is a scalar integer expression identifying a data transfer operation that was returned using the ID= specifier in a previous asynchronous READ or WRITE statement.

This specifier is used with the PENDING specifier to determine whether a specific asynchronous pending data transfer is completed.

See Also

- INQUIRE Statement Overview
- PENDING specifier

INQUIRE: IOFOCUS Specifier (W*32, W*64)

The IOFOCUS specifier asks if the indicated unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof	Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified unit is the active window in a QuickWin application
.FALSE.	If the specified unit is not the active window in a QuickWin application

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any READ, WRITE, or PRINT statement to that window.

If you use this specifier with a non-Windows application, an error occurs.

INQUIRE: MODE Specifier

MODE is a nonstandard synonym for ACTION.

INQUIRE: NAME Specifier

The NAME specifier returns the name of a file. It takes the following form:

NAME = *nme*

nme	Is a scalar default character variable that is assigned the name of the file to which the unit is connected. If the file does not have a name, *nme* is undefined. The value assigned to *nme* is not necessarily the same as the value given in the FILE specifier. However, the value that is assigned is always valid for use with the FILE specifier in an OPEN statement, unless the value has been truncated in a way that makes it unacceptable. (Values are truncated if the declaration of *nme* is too small to contain the entire value.)

NOTE. The FILE and NAME specifiers are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

See Also

- INQUIRE Statement Overview

The appropriate manual in your operating system documentation set for details on the maximum size of file pathnames

INQUIRE: NAMED Specifier

The NAMED specifier asks whether a file is named. It takes the following form:

NAMED = *nmd*

nmd	Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the file has a name

.FALSE.	If the file does not have a name

INQUIRE: NEXTREC Specifier

The NEXTREC specifier asks where the next record can be read or written in a file connected for direct access. It takes the following form:

NEXTREC = *nr*

nr Is a scalar integer variable that is assigned a value as follows:

- If the file is connected for direct access and a record (r) was previously read or written, the value assigned is r + 1.
- If no record has been read or written, the value assigned is 1.
- If the file is not connected for direct access, or if the file position cannot be determined because of an error condition, the value assigned is zero.
- If the file is connected for direct access and a REWIND has been performed on the file, the value assigned is 1.

INQUIRE: NUMBER Specifier

The NUMBER specifier asks the number of the unit connected to a file. It takes the following form:

NUMBER = *num*

num Is a scalar integer variable.
The *num* is assigned the number of the unit currently connected to the specified file. If there is no unit connected to the file, the value assigned is -1.

INQUIRE: OPENED Specifier

The OPENED specifier asks whether a file is connected. It takes the following form:

OPENED = *od*

od Is a scalar default logical variable that is assigned one of the following values:

.TRUE.	If the specified file or unit is connected

.FALSE.	If the specified file or unit is not connected

INQUIRE: ORGANIZATION Specifier

The ORGANIZATION specifier asks how the file is organized. It takes the following form:

ORGANIZATION = *org*

org Is a scalar default character variable that is assigned one of the following values:

'SEQUENTIAL'	If the file is a sequential file
'RELATIVE'	If the file is a relative file
'UNKNOWN'	If the processor cannot determine the file's organization

INQUIRE: PAD Specifier

The PAD specifier asks whether blank padding was specified for the file. It takes the following form:

PAD = *pd*

pd Is a scalar default character variable that is assigned one of the following values:

'NO'	If the file or unit was connected with PAD='NO'
'YES'	If the file or unit is not connected, or it was connected with PAD='YES'

INQUIRE: PENDING Specifier

The PENDING specifier asks whether previously pending asynchronous data transfers are complete. A data transfer is previously pending if it is not complete at the beginning of execution of the INQUIRE statement. It takes the following form:

PENDING = *pnd*

pnd Is a scalar default logical variable that is assigned the value .TRUE. or .FALSE..

The value is assigned as follows:

- If an ID specifier appears in the INQUIRE statement, the following occurs:
  - If the data transfer specified by ID is complete, then variable *pnd* is set to .FALSE. and INQUIRE performs the WAIT operation for the specified data transfer.
  - If the data transfer specified by ID is not complete, then variable *pnd* is set to .TRUE. and no WAIT operation is performed. The previously pending data transfer remains pending after the execution of the INQUIRE statement.
- If an ID specifier does not appear in the INQUIRE statement, the following occurs:
  - If all previously pending data transfers for the specified unit are complete, then variable *pnd* is set to .FALSE. and the INQUIRE statement performs WAIT operations for all previously pending data transfers for the specified unit.
  - If there are data transfers for the specified unit that are not complete, then variable *pnd* is set to .TRUE. and no WAIT operations are performed. The previously pending data transfers remain pending after the execution of the INQUIRE statement.

See Also

- INQUIRE Statement Overview
- INQUIRE: ID Specifier
- Example in INQUIRE Statement

INQUIRE: POS Specifier

The POS specifier identifies the file position in file storage units in a stream file. It takes the following form:

POS = *p*

p Is a scalar integer variable that is assigned the number of the file storage unit immediately following the current position of a file connected for stream access (ACCESS='STREAM').

If the file is positioned at its terminal position, p is assigned a value one greater than the number of the highest-numbered file storage unit in the file.

If the file is not connected for stream access or if the position of the file is indeterminate because of previous error conditions, *p* is assigned the value one.

INQUIRE: POSITION Specifier

The POSITION specifier asks the position of the file. It takes the following form:

POSITION = *pos*

pos Is a scalar default character variable that is assigned one of the following values:

'REWIND'	If the file is connected with its position at its initial point
'APPEND'	If the file is connected with its position at its terminal point (or before its end-of-file record, if any)
'ASIS'	If the file is connected without changing its position
'UNDEFINED'	If the file is not connected, or is connected for direct access data transfer and a REWIND statement has not been performed on the unit

See Also

- INQUIRE Statement Overview

Building Applications for details on record position, advancement, and transfer

INQUIRE: READ Specifier

The READ specifier asks whether a file can be read. It takes the following form:

READ = *rd*

rd Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be read
'NO'	If the file cannot be read

'UNKNOWN'	If the processor cannot determine whether the file can be read

INQUIRE: READWRITE Specifier

The READWRITE specifier asks whether a file can be both read and written to. It takes the following form:

READWRITE = *rdwr*

rdwr Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be both read and written to
'NO'	If the file cannot be both read and written to
'UNKNOWN'	If the processor cannot determine whether the file can be both read and written to

INQUIRE: RECL Specifier

The RECL specifier asks the maximum record length for a file. It takes the following form:

RECL = *rcl*

rcl Is a scalar integer variable that is assigned a value as follows:

- If the file or unit is connected, the value assigned is the maximum record length allowed.
- If the file does not exist, or is not connected, the value assigned is zero.

The assigned value is expressed in 4-byte units if the file is currently (or was previously) connected for unformatted data transfer; otherwise, the value is expressed in bytes.

INQUIRE: RECORDTYPE Specifier

The RECORDTYPE specifier asks which type of records are in a file. It takes the following form:

RECORDTYPE = *rtype*

rtype	Is a scalar default character variable that is assigned one of the following values:	
	'FIXED'	If the file is connected for fixed-length records
	'VARIABLE'	If the file is connected for variable-length records
	'SEGMENTED'	If the file is connected for unformatted sequential data transfer using segmented records
	'STREAM'	If the file's records are not terminated
	'STREAM_CR'	If the file's records are terminated with a carriage return
	'STREAM_LF'	If the file's records are terminated with a line feed
	'STREAM_CRLF'	If the file's records are terminated with a carriage return/line feed pair
	'UNKNOWN'	If the file is not connected

INQUIRE: SEQUENTIAL Specifier

The SEQUENTIAL specifier asks whether a file is connected for sequential access. It takes the following form:

SEQUENTIAL = *seq*

seq	Is a scalar default character variable that is assigned one of the following values:	
	'YES'	If the file is connected for sequential access
	'NO'	If the file is not connected for sequential access

'UNKNOWN'	If the processor cannot determine whether the file is connected for sequential access

INQUIRE: SHARE Specifier

The SHARE specifier asks the current share status of a file or unit. It takes the following form:

SHARE = *shr*

shr Is a scalar default character variable.
On Windows* systems, this variable is assigned one of the following values:

'DENYRW'	If the file is connected for deny-read/write mode
'DENYWR'	If the file is connected for deny-write mode
'DENYRD'	If the file is connected for deny-read mode
'DENYNONE'	If the file is connected for deny-none mode
'UNKNOWN'	If the file or unit is not connected

On Linux* and Mac OS* X systems, this variable is assigned one of the following values:

'DENYRW'	If the file is connected for exclusive access
'DENYNONE'	If the file is connected for shared access
'NODENY'	If the file is connected with default locking
'UNKNOWN'	If the file or unit is not connected

INQUIRE: UNFORMATTED Specifier

The UNFORMATTED specifier asks whether a file is connected for unformatted data transfer. It takes the following form:

UNFORMATTED = *unf*

unf Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file is connected for unformatted data transfer
'NO'	If the file is not connected for unformatted data transfer
'UNKNOWN'	If the processor cannot determine whether the file is connected for unformatted data transfer

INQUIRE: WRITE Specifier

The WRITE specifier asks whether a file can be written to. It takes the following form:

WRITE = *wr*

wr Is a scalar default character variable that is assigned one of the following values:

'YES'	If the file can be written to
'NO'	If the file cannot be written to
'UNKNOWN'	If the processor cannot determine whether the file can be written to

OPEN Statement Overview

The OPEN statement connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection. For more information, see OPEN.

The following table summarizes the OPEN statement specifiers and their values (and contains links to their descriptions):

Table 652: OPEN Statement Specifiers and Values

Specifier	Values	Function	Default
ACCESS	'SEQUENTIAL' 'DIRECT' 'APPEND'	Access mode	'SEQUENTIAL'
ACTION (or MODE)	'READ' 'WRITE' 'READWRITE'	File access	'READWRITE'
ASSOCIATEVARIABLE	var	Next direct access record	No default
BLANK	'NULL' 'ZERO'	Interpretation of blanks	'NULL'
BLOCKSIZE	n_expr	Physical block size	Filesystem default
BUFFERCOUNT	n_expr	Number of I/O buffers	One
BUFFERED	'YES' 'NO'	Buffering for WRITE operations	'NO'
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Print control	Formatted: 'LIST'[1] Unformatted: 'NONE'
CONVERT	'LITTLE_ENDIAN' 'BIG_ENDIAN' 'CRAY' 'FDX' 'FGX' 'IBM'	Numeric format specification	'NATIVE'

Specifier	Values	Function	Default
	'VAXD' 'VAXG' 'NATIVE'		
DEFAULTFILE	c_expr	Default file pathname	Current working directory
DELIM	'APOSTROPHE' 'QUOTE' 'NONE'	Delimiter for character constants	'NONE'
DISPOSE (or DISP)	'KEEP' or 'SAVE' 'DELETE' 'PRINT' 'PRINT/DELETE' 'SUBMIT' 'SUBMIT/DELETE'	File disposition at close	'KEEP'
ERR	label	Error transfer control	No default
FILE (or NAME)	c_expr	File pathname (file name)	fort.n [2]
FORM	'FORMATTED' 'UNFORMATTED' 'BINARY'	Format type	Depends on ACCESS setting
IOFOCUS	.TRUE. .FALSE.	Active window in QuickWin application	.TRUE.[3]
IOSTAT	var	I/O status	No default
MAXREC	n_expr	Direct access record limit	No limit
NOSHARED [4]	No value	File sharing disallowed	L*X, M*X: SHARED

Specifier	Values	Function	Default
			W*32, W*64: Not shared
ORGANIZATION	'SEQUENTIAL' 'RELATIVE'	File organization	'SEQUENTIAL'
PAD	'YES' 'NO'	Record padding	'YES'
POSITION	'ASIS' 'REWIND' 'APPEND'	File positioning	'ASIS'
READONLY	No value	Write protection	No default
RECL (or RECORDSIZE)	n_expr	Record length	Depends on RECORDTYPE, ORGANIZATION, and FORM settings [5]
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED' 'STREAM' 'STREAM_CR' 'STREAM_LF'	Record type	Depends on ORGANIZATION, CARRIAGECONTROL, ACCESS, and FORM settings
SHARE [4]	'DENYRW' 'DENYWR'[6] 'DENYRD'[6] 'DENYNONE'	File locking	'DENYWR' [7]
SHARED [4]	No value	File sharing allowed	L*X, M*X: SHARED W*32, W*64: Not shared

Specifier	Values	Function	Default
STATUS (or TYPE)	'OLD' 'NEW' 'SCRATCH' 'REPLACE' 'UNKNOWN'	File status at open	'UNKNOWN' [8]
TITLE	c_expr	Title for child window in QuickWin application	No default
UNIT	n_expr	Logical unit number	No default; an io-unit must be specified
USEROPEN	func	User program option	No default

[1] If you use the compiler option specifying OpenVMS defaults, and the unit is connected to a terminal, the default is 'FORTRAN'.

[2] n is the unit number.

[3] If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

[4] For information on file sharing, see *Building Applications*.

[5] On Linux* and Mac OS* X systems, the default depends only on the FORM setting.

[6] W*32, W*64

[7] The default differs under certain conditions (see SHARE Specifier).

[8] The default differs under certain conditions (see STATUS Specifier).

Key to Values

```
c_expr: A scalar default character expression
func:   An external function
label:  A statement label
n_expr: A scalar numeric expression
var:    A scalar integer variable
```

OPEN: ACCESS Specifier

The ACCESS specifier indicates the access method for the connection of the file. It takes the following form:

ACCESS = *acc*

acc Is a scalar default character expression that evaluates to one of the following values:

'DIRECT'	Indicates direct access.
'SEQUENTIAL'	Indicates sequential access.
'STREAM'	Indicates stream access, where the file storage units of the file are accessible sequentially or by position.
'APPEND'	Indicates sequential access, but the file is positioned at the end-of-file record.

The default is 'SEQUENTIAL'.

There are limitations on record access by file organization and record type. For more information, see *Building Applications*.

OPEN: ACTION Specifier

The ACTION specifier indicates the allowed I/O operations for the file connection. It takes the following form:

ACTION = *act*

act Is a scalar default character expression that evaluates to one of the following values:

'READ'	Indicates that only READ statements can refer to this connection.
'WRITE'	Indicates that only WRITE, DELETE, and ENDFILE statements can refer to this connection.

'READWRITE'	Indicates that READ, WRITE, DELETE, and ENDFILE statements can refer to this connection.

The default is 'READWRITE'.

However, if compiler option fpscomp general is specified on the command line and *action* is omitted, the system first attempts to open the file with 'READWRITE'. If this fails, the system tries to open the file again, first using 'READ', then using 'WRITE'.

Note that in this case, omitting *action* is not the same as specifying ACTION='READWRITE'. If you specify ACTION='READWRITE' and the file cannot be opened for both read and write access, the attempt to open the file fails. You can use the INQUIRE statement to determine the actual access mode selected.

See Also

- OPEN Statement Overview
- fpscomp compiler option

OPEN: ASSOCIATEVARIABLE Specifier

The ASSOCIATEVARIABLE specifier indicates a variable that is updated after each direct access I/O operation, to reflect the record number of the next sequential record in the file. It takes the following form:

ASSOCIATEVARIABLE = *asv*

asv	Is a scalar integer variable. It cannot be a dummy argument to the routine in which the OPEN statement appears.

Direct access READs, direct access WRITEs, and the FIND, DELETE, and REWRITE statements can affect the value of *asv*.

This specifier is valid only for direct access; it is ignored for other access modes.

OPEN: ASYNCHRONOUS Specifier

The ASYNCHRONOUS specifier indicates whether asynchronous I/O is allowed for a unit. It takes the following form:

ASYNCHRONOUS = *asyn*

asyn	Is a scalar expression of type default character that evaluates to one of the following values:

'YES'	Indicates that asynchronous I/O is allowed for a unit.
'NO'	Indicates that asynchronous I/O is not allowed for a unit.

The default is 'NO'.

OPEN: BLANK Specifier

The BLANK specifier indicates how blanks are interpreted in a file. It takes the following form:

BLANK = *blnk*

blnk Is a scalar default character expression that evaluates to one of the following values:

'NULL'	Indicates all blanks are ignored, except for an all-blank field (which has a value of zero).
'ZERO'	Indicates all blanks (other than leading blanks) are treated as zeros.

The default is 'NULL' (for explicitly OPENed files, preconnected files, and internal files). If you specify compiler option f66 (or OPTIONS/NOF77), the default is 'ZERO'.

If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks.

See Also

- OPEN Statement Overview
- Blank Editing
- f66

OPEN: BLOCKSIZE Specifier

The BLOCKSIZE specifier indicates the physical I/O transfer size for the file. It takes the following form:

BLOCKSIZE = *bks*

bks	Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

If you specify a nonzero number for *bks*, it is rounded up to a multiple of 512 byte blocks.

If you do not specify BLOCKSIZE or you specify zero for *bks*, the filesystem default value is assumed.

Note that blocksize is meaningful for sequential access only.

OPEN: BUFFERCOUNT Specifier

The BUFFERCOUNT specifier indicates the number of buffers to be associated with the unit for multibuffered I/O. It takes the following form:

BUFFERCOUNT = *bc*

bc	Is a scalar numeric expression in the range 1 through 127. If necessary, the value is converted to integer data type before use.

The BLOCKSIZE specifier determines the size of each buffer. For example, if BUFFERCOUNT=3 and BLOCKSIZE=2048, the total number of bytes allocated for buffers is 3*2048, or 6144 bytes.

If you do not specify BUFFERCOUNT or you specify zero for *bc*, the default is 1.

See Also

- OPEN Statement Overview
- BLOCKSIZE specifier

Optimizing Applications for details on obtaining optimal run-time performance

OPEN: BUFFERED Specifier

The BUFFERED specifier indicates run-time library behavior following WRITE operations. It takes the following form:

BUFFERED = *bf*

bf	Is a scalar default character expression that evaluates to one of the following values:	
	'NO'	Requests that the run-time library send output data to the file system after each WRITE operation.

'YES'	Requests that the run-time library accumulate output data in its internal buffer, possibly across several WRITE operations, before the data is sent to the file system. Buffering may improve run-time performance for output-intensive applications.

The default is 'NO'.

If BUFFERED='YES' is specified, the request may or may not be honored, depending on the output device and other file or connection characteristics.

For direct access, you should specify BUFFERED='YES', although using direct-access I/O to a network file system may be much slower.

If BLOCKSIZE and BUFFERCOUNT have been specified for OPEN, their product determines the size in bytes of the internal buffer. Otherwise, the default size of the internal buffer is 8192 bytes.

NOTE. On Windows systems, the default size of the internal buffer is 1024 bytes if compiler option fpscomp general is used.

The internal buffer will grow to hold the largest single record but will never shrink.

OPEN: CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier indicates the type of carriage control used when a file is displayed at a terminal. It takes the following form:

CARRIAGECONTROL = *cc*

cc Is a scalar default character expression that evaluates to one of the following values:

'FORTRAN'	Indicates normal Fortran interpretation of the first character.
'LIST'	Indicates one line feed between records.

'NONE'	Indicates no carriage control processing.

The default for binary (W*32, W*64) and unformatted files is 'NONE'. The default for formatted files is 'LIST'. However, if you specify compiler option vms or fpscomp general, and the unit is connected to a terminal, the default is 'FORTRAN'.

On output, if a file was opened with CARRIAGECONTROL='FORTRAN' in effect or the file was processed by the `fortpr` format utility, the first character of a record transmitted to a line printer or terminal is typically a character that is not printed, but is used to control vertical spacing.

See Also

- OPEN Statement Overview
- Printing of Formatted Records
- vms
- fpscomp

OPEN: CONVERT Specifier

The CONVERT specifier indicates a nonnative numeric format for unformatted data. It takes the following form:

CONVERT = *fm*

fm Is a scalar default character expression that evaluates to one of the following values:

'LITTLE_ENDIAN'[1]	Little endian integer data [2] and IEEE* floating-point data. [3]
'BIG_ENDIAN' [1]	Big endian integer data [2] and IEEE floating-point data. [3]
'CRAY'	Big endian integer data [2] and CRAY* floating-point data of size REAL(8) or COMPLEX(8).
'FDX'	Little endian integer data [2] and VAX* processor floating-point data of format F_floating for REAL(4) or COMPLEX(4),

	D_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'FGX'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and IEEE X_floating for REAL(16) or COMPLEX(16).
'IBM'	Big endian integer data [2] and IBM* System\370 floating-point data of size REAL(4) or COMPLEX(4) (IBM short 4), and size REAL(8) or COMPLEX(8) (IBM long 8).
'VAXD'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), D_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'VAXG'	Little endian integer data [2] and VAX processor floating-point data of format F_floating for size REAL(4) or COMPLEX(4), G_floating for size REAL(8) or COMPLEX(8), and H_floating for REAL(16) or COMPLEX(16).
'NATIVE'	No data conversion. This is the default.

[1] INTEGER(1) data is the same for little endian and big endian.

[2] Of the appropriate size: INTEGER(1), INTEGER(2), INTEGER(4), or INTEGER(8)

[3] Of the appropriate size and type: REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), or COMPLEX(16)

You can use CONVERT to specify multiple formats in a single program, usually one format for each specified unit number.

When reading a nonnative format, the nonnative format on disk is converted to native format in memory. If a converted nonnative value is outside the range of the native data type, a run-time message appears.

There are other ways to specify numeric format for unformatted files: you can specify an environment variable, compiler option convert, or OPTIONS/CONVERT. The following shows the order of precedence:

Method Used	Precedence
An environment variable	Highest
OPEN (CONVERT=)	.
OPTIONS/CONVERT	.
The convert compiler option	Lowest

Compiler option convert and OPTIONS/CONVERT affect all unit numbers used by the program, while environment variables and OPEN (CONVERT=) affect specific unit numbers.

The following example shows how to code the OPEN statement to read unformatted CRAY* numeric data from unit 15, which might be processed and possibly written in native little endian format to unit 20:

```
   OPEN (CONVERT='CRAY', FILE='graph3.dat', FORM='UNFORMATTED',
1      UNIT=15)
   ...
   OPEN (FILE='graph3_native.dat', FORM='UNFORMATTED', UNIT=20)
```

See Also

- OPEN Statement Overview
- Data Types, Constants, and Variables
- convert compiler option

Building Applications: Setting Environment Variables

Building Applications: Run-Time Environment Variables

Building Applications for details on supported ranges for data types

OPEN: DEFAULTFILE Specifier

The DEFAULTFILE specifier indicates a default file pathname string. It takes the following form:

DEFAULTFILE = *def*

def	Is a character expression indicating a default file pathname string. The default file pathname string is used primarily when accepting file pathnames interactively. File pathnames known to a user program normally appear in the FILE specifier.

DEFAULTFILE supplies a value to the Fortran I/O system that is prefixed to the name that appears in FILE.

If *def* does not end in a slash (/), a slash is added.

If DEFAULTFILE is omitted, the Fortran I/O system uses the current working directory.

OPEN: DELIM Specifier

The DELIM specifier indicates what characters (if any) are used to delimit character constants in list-directed and namelist output. It takes the following form:

DELIM = *del*

del Is a scalar default character expression that evaluates to one of the following values:

'APOSTROPHE'	Indicates apostrophes delimit character constants. All internal apostrophes are doubled.
'QUOTE'	Indicates quotation marks delimit character constants. All internal quotation marks are doubled.
'NONE'	Indicates character constants have no delimiters. No internal apostrophes or quotation marks are doubled.

The default is 'NONE'.

The DELIM specifier is only allowed for files connected for formatted data transfer; it is ignored during input.

OPEN: DISPOSE Specifier

The DISPOSE (or DISP) specifier indicates the status of the file after the unit is closed. It takes one of the following forms:

DISPOSE = *dis*

DISP = *dis*

dis Is a scalar default character expression that evaluates to one of the following values:

'KEEP' or 'SAVE'	Retains the file after the unit closes.
'DELETE'	Deletes the file after the unit closes.
'PRINT' [1]	Submits the file to the line printer spooler and retains it.
'PRINT/DELETE' [1]	Submits the file to the line printer spooler and then deletes it.
'SUBMIT'	Forks a process to execute the file.
'SUBMIT/DELETE'	Forks a process to execute the file, and then deletes the file after the fork is completed.

[1] Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

[1] Use only on sequential files.

The default is 'DELETE' for scratch files. For all other files, the default is 'KEEP'.

OPEN: FILE Specifier

The FILE specifier indicates the name of the file to be connected to the unit. It takes the following form:

FILE = *name*

name	Is a character or numeric expression. The *name* can be any pathname allowed by the operating system. Any trailing blanks in the name are ignored.

If the following conditions occur:

- FILE is omitted
- The unit is not connected to a file
- STATUS='SCRATCH' is not specified
- The corresponding FORT*n* environment variable is not set for the unit number

then Intel® Fortran generates a file name in the form fort.*n*, where *n* is the logical unit number. On Windows systems, if compiler option fpscomp general is specified, omitting FILE implies STATUS='SCRATCH'.

If the file name is stored in a numeric scalar or array, the name must consist of ASCII characters terminated by an ASCII null character (zero byte). However, if it is stored in a character scalar or array, it must not contain a zero byte.

On Windows systems, if the filename is 'USER' or 'CON', input and output are directed to the console. For a complete list of device names, see Physical Devices in *Building Applications*.

In a Windows* QuickWin application, you can specify FILE='USER' to open a child window. All subsequent I/O statements directed to that unit appear in the child window.

On Windows systems, the *name* can be blank (FILE=' ') if the compatibility compiler option fpscomp filesfromcmd is specified. If the *name* is blank, the following occurs:

1. The program reads a filename from the list of arguments (if any) in the command line that started the program. If the argument is a null or blank string (" "), you are prompted for the corresponding filename. Each successive OPEN statement that specifies a blank name reads the next following command-line argument.
2. If no command-line arguments are specified or there are no more arguments in the list, you are prompted for additional filenames.

   Assume the following command line started the program MYPROG (note that quotation marks (") are used):

   myprog first.fil " " third.txt

   MYPROG contains four OPEN statements with blank filenames, in the following order:

```
OPEN (2, FILE = ' ')

OPEN (4, FILE = ' ')

OPEN (5, FILE = ' ')

OPEN (10, FILE = ' ')
```

Unit 2 is associated with the file FIRST.FIL. Because a blank argument was specified on the command line for the second filename, the OPEN statement for unit 4 produces the following prompt:

```
Filename missing or blank - Please enter name UNIT 4?
```

Unit 5 is associated with the file THIRD.TXT. Because no fourth file was specified on the command line, the OPEN statement for unit 10 produces the following prompt:

```
Filename missing or blank - Please enter name UNIT 10?
```

See Also

- OPEN Statement Overview
- fpscomp compiler option

Building Applications: Physical Devices

Building Applications for details on default file name conventions

The appropriate manual in your system documentation set for details on allowable file pathnames

OPEN: FORM Specifier

The FORM specifier indicates whether the file is being connected for formatted, unformatted, or binary (W*32, W*64) data transfer. It takes the following form:

FORM = *fm*

fm Is a scalar default character expression that evaluates to one of the following values:

'FORMATTED'	Indicates formatted data transfer
'UNFORMATTED'	Indicates unformatted data transfer
'BINARY'	Indicates binary data transfer

The default is 'FORMATTED' for sequential access files, and 'UNFORMATTED' for direct access files.

The data is stored and retrieved in a file according to the file's access (set by the ACCESS specifier) and the form of the data the file contains.

A *formatted file* is a sequence of formatted records. Formatted records are a series of ASCII characters terminated by an end-of-record mark (a carriage return and line feed sequence). The records in a formatted direct-access file must all be the same length. The records in a formatted sequential file can have varying lengths. All internal files must be formatted.

An *unformatted file* is a sequence of unformatted records. An unformatted record is a sequence of values. Unformatted direct files contain only this data, and each record is padded to a fixed length with undefined bytes. Unformatted sequential files contain the data plus information that indicates the boundaries of each record.

Binary sequential files are sequences of bytes with no internal structure. There are no records. The file contains only the information specified as I/O list items in WRITE statements referring to the file.

Binary direct files have very little structure. A record length is assigned by the RECL specifier in an OPEN statement. This establishes record boundaries, which are used only for repositioning and padding before and after read and write operations and during BACKSPACE operations. Record boundaries do not restrict the number of bytes that can be transferred during a read or write operation. If an I/O operation attempts to read or write more values than are contained in a record, the read or write operation is continued on the next record.

OPEN: IOFOCUS Specifier (W*32, W*64)

The IOFOCUS specifier indicates whether a particular unit is the active window in a QuickWin application. It takes the following form:

IOFOCUS = *iof*

iof Is a scalar default logical expression that evaluates to one of the following values:

.TRUE.	Indicates the QuickWin child window is the active window
.FALSE.	Indicates the QuickWin child window is not the active window

If unit '*' is specified, the default is .FALSE.; otherwise, the default is .TRUE..

A value of .TRUE. causes a call to FOCUSQQ immediately before any READ, WRITE, or PRINT statement to that window. OUTTEXT, OUTGTEXT, or any other graphics routine call does not cause the focus to shift.

See Also

- OPEN Statement Overview

Building Applications: Giving a Window Focus and Setting the Active Window

OPEN: MAXREC Specifier

The MAXREC specifier indicates the maximum number of records that can be transferred from or to a direct access file while the file is connected. It takes the following form:

MAXREC = *mr*

mr	Is a scalar numeric expression. If necessary, the value is converted to integer data type before use.

The default is an unlimited number of records.

OPEN: MODE Specifier

MODE is a nonstandard synonym for ACTION.

OPEN: NAME Specifier

NAME is a nonstandard synonym for FILE.

OPEN: NOSHARED Specifier

The NOSHARED specifier indicates that the file is connected for exclusive access by the program. It takes the following form:

NOSHARED

See Also

- OPEN Statement Overview

Building Applications for details on file sharing

OPEN: ORGANIZATION Specifier

The ORGANIZATION specifier indicates the internal organization of the file. It takes the following form:

ORGANIZATION = *org*

org	Is a scalar default character expression that evaluates to one of the following values	
	'SEQUENTIAL'	Indicates a sequential file.
	'RELATIVE'	Indicates a relative file.

The default is 'SEQUENTIAL'.

OPEN: PAD Specifier

The PAD specifier indicates whether a formatted input record is padded with blanks when an input list and format specification requires more data than the record contains.

The PAD specifier takes the following form:

PAD = *pd*

pd Is a scalar default character expression that evaluates to one of the following values:

'YES'	Indicates the record will be padded with blanks when necessary.
'NO'	Indicates the record will not be padded with blanks. The input record must contain the data required by the input list and format specification.

The default is 'YES'.

This behavior is different from FORTRAN 77, which never pads short records with blanks. For example, consider the following:

```
READ (5,'(I5)') J
```

If you enter 123 followed by a carriage return, FORTRAN 77 turns the I5 into an I3 and J is assigned 123.

However, Intel Fortran pads the 123 with 2 blanks unless you explicitly open the unit with PAD='NO'.

You can override blank padding by explicitly specifying the BN edit descriptor.

The PAD specifier is ignored during output.

OPEN: POSITION Specifier

The POSITION specifier indicates the position of a file connected for sequential access. It takes the following form:

POSITION = *pos*

pos Is a scalar default character expression that evaluates to one of the following values:

'ASIS'	Indicates the file position is unchanged if the file exists and is already connected. The position is unspecified if the file exists but is not connected.
'REWIND'	Indicates the file is positioned at its initial point.
'APPEND'	Indicates the file is positioned at its terminal point (or before its end-of-file record, if any).

The default is 'ASIS'. (On Fortran I/O systems, this is the same as 'REWIND'.)

A new file (whether specified as new explicitly or by default) is always positioned at its initial point.

In addition to the POSITION specifier, you can use position statements. The BACKSPACE statement positions a file back one record. The REWIND statement positions a file at its initial point. The ENDFILE statement writes an end-of-file record at the current position and positions the file after it. Note that ENDFILE does not go the end of an existing file, but creates an end-of-file where it is.

See Also

- OPEN Statement Overview

Building Applications for details on record position, advancement, and transfer

OPEN: READONLY Specifier

The READONLY specifier indicates only READ statements can refer to this connection. It takes the following form:

READONLY

READONLY is similar to specifying ACTION='READ', but READONLY prevents deletion of the file if it is closed with STATUS='DELETE' in effect.

The Fortran I/O system's default privileges for file access are READWRITE. If access is denied, the I/O system automatically retries accessing the file for READ access.

However, if you use compiler option vms, the I/O system does not retry accessing for READ access. So, run-time I/O errors can occur if the file protection does not permit WRITE access. To prevent such errors, if you wish to read a file for which you do not have write access, specify READONLY.

OPEN: RECL Specifier

The RECL specifier indicates the length of each record in a file connected for direct access, or the maximum length of a record in a file connected for sequential access.

The RECL specifier takes the following form:

RECL = *rl*

rl	Is a positive numeric expression indicating the length of records in the file. If necessary, the value is converted to integer data type before use.

If the file is connected for formatted data transfer, the value must be expressed in bytes (characters). Otherwise, the value is expressed in 4-byte units (longwords). If the file is connected for unformatted data transfer, the value can be expressed in bytes if compiler option assume byterecl is specified.

Except for segmented records, the *rl* is the length for record data only, it does not include space for control information. If *rl* is too large, you can exhaust your program's virtual memory resources trying to create room for the record.

The length specified is interpreted depending on the type of records in the connected file, as follows:

- For segmented records, RECL indicates the maximum length for any segment (including the four bytes of control information).
- For fixed-length records, RECL indicates the size of each record; it *must* be specified. If the records are unformatted, the size must be expressed as an even multiple of four.

  You can use the RECL specifier in an INQUIRE statement to get the record length before opening the file.
- For variable-length records, RECL indicates the maximum length for any record.

If you read a fixed-length file with a record length different from the one used to create the file, indeterminate results can occur.

The maximum length for *rl* depends on the record type and the setting of the CARRIAGECONTROL specifier, as shown in the following table:

Table 668: Maximum Record Lengths (RECL)

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Fixed-length	'NONE'	2147483647 (2**31-1) [1]
Variable-length	'NONE'	2147483640 (2**31-8)

Record Type	CARRIAGECONTROL	Formatted (size in bytes)
Segmented	'NONE'	32764 (2**15-4)
Stream	'NONE'	2147483647 (2**31-1)
Stream_CR	'LIST'	2147483647 (2**31-1)
	'FORTRAN'	2147483646 (2**31-2)
Stream_LF	'LIST'	2147483647 (2**31-1) [2]
	'FORTRAN'	2147483646 (2**31-2)

[1]Subtract 1 if compiler option vms is used.

[2] L*X only

The default value depends on the setting of the RECORDTYPE specifier, as shown in the following table:

Table 669: Default Record Lengths (RECL)

RECORDTYPE	RECL value
'FIXED'	None; value must be explicitly specified.
All other settings	132 bytes for formatted records; 510 longwords for unformatted records.[1]

[1]To change the default record length values, you can use environment variable FORT_FMT_RECL or FORT_UFMT_RECL.

For formatted records with other than RECORDTYPE='FIXED', the default RECL is 132. For example, you can write record lengths of 132 characters without a "maximum record length exceeded" error. However, after 80 characters, the remaining characters will wrap to the next line. Therefore, writing 100 characters will produce two lines of output. This is a property of terminal format files called the *right margin*. If you do not specify a RECL, this right margin is set to 80; otherwise, it is set to the value of the RECL.

See Also

- OPEN Statement Overview
- assume byterecl compiler option
- vms compiler option

OPEN: RECORDSIZE Specifier

RECORDSIZE is a nonstandard synonym for RECL.

OPEN: RECORDTYPE Specifier

The RECORDTYPE specifier indicates the type of records in a file. It takes the following form:

RECORDTYPE = *typ*

typ Is a scalar default character expression that evaluates to one of the following values:

'FIXED'	Indicates fixed-length records.
'VARIABLE'	Indicates variable-length records.
'SEGMENTED'	Indicates segmented records.
'STREAM'	Indicates stream-type variable length data with no record terminators.
'STREAM_LF'	Indicates stream-type variable length records, terminated with a line feed.
'STREAM_CR'	Indicates stream-type variable length records, terminated with a carriage return.
'STREAM_CRLF'	Indicates stream-type variable length records, terminated with a carriage return/line feed pair.

When you open a file, default record types are as follows:

'FIXED'	For relative files
'FIXED'	For direct access sequential files
'STREAM_LF'	For formatted sequential access files on Linux* and Mac OS* X systems

'STREAM_CRLF'	For formatted sequential access files on Windows systems
'VARIABLE'	For unformatted sequential access files

A *segmented record* is a logical record consisting of segments that are physical records. Since the length of a segmented record can be greater than 65,535 bytes, only use segmented records for unformatted sequential access to disk or raw magnetic tape files.

Files containing segmented records can be accessed only by unformatted sequential data transfer statements.

If an output statement does not specify a full record for a file containing fixed-length records, the following occurs:

- In formatted files, the record is filled with blanks
- In unformatted files, the record is filled with zeros

See Also

- OPEN Statement Overview

Building Applications for details on record types and file organization

OPEN: SHARE Specifier

The SHARE specifier indicates whether file locking is implemented while the unit is open. It takes the following form:

SHARE = *shr*

shr Is a scalar default character expression.
On Windows* systems, this expression evaluates to one of the following values:

'DENYRW'	Indicates deny-read/write mode. No other process can open the file.
'DENYWR'	Indicates deny-write mode. No process can open the file with write access.
'DENYRD'	Indicates deny-read mode. No process can open the file with read access.

'DENYNONE'	Indicates deny-none mode. Any process can open the file in any mode.
On Linux* and Mac OS* X systems, this expression evaluates to one of the following values:	
'DENYRW'	Indicates exclusive access for cooperating processes.
'DENYNONE'	Indicates shared access for cooperating processes.

On Windows systems, the default is 'DENYWR'. However, if you specify compiler option fpscomp general or the SHARED specifier, the default is 'DENYNONE'.

On Linux and Mac OS X systems, no restrictions are applied to file opening if you do not use a locking mechanism.

'COMPAT' is accepted for compatibility with previous versions. It is equivalent to 'DENYNONE'.

Use the ACCESS specifier in an INQUIRE statement to determine the access permission for a file.

Be careful not to permit other users to perform operations that might cause problems. For example, if you open a file intending only to read from it, and want no other user to write to it while you have it open, you could open it with ACTION='READ' and SHARE='DENYRW'. Other users would not be able to open it with ACTION='WRITE' and change the file.

Suppose you want several users to read a file, and you want to make sure no user updates the file while anyone is reading it. First, determine what type of access to the file you want to allow the original user. Because you want the initial user to read the file only, that user should open the file with ACTION='READ'. Next, determine what type of access the initial user should allow other users; in this case, other users should be able only to read the file. The first user should open the file with SHARE='DENYWR'. Other users can also open the same file with ACTION='READ' and SHARE='DENYWR'.

See Also

- OPEN Statement Overview
- fpscomp compiler option

Building Applications for details about limitations on record access

OPEN: SHARED Specifier

The SHARED specifier indicates that the file is connected for shared access by more than one program executing simultaneously. It takes the following form:

SHARED

On Linux* and Mac OS* X systems, shared access is the default for the Fortran I/O system. On Windows* systems, it is the default if SHARED or compiler option fpscomp general is specified.

See Also

- OPEN Statement Overview
- fpscomp general compiler option

Building Applications for details on file sharing

OPEN: STATUS Specifier

The STATUS specifier indicates the status of a file when it is opened. It takes the following form:

STATUS = *sta*

sta Is a scalar default character expression that evaluates to one of the following values:

'OLD'	Indicates an existing file.
'NEW'	Indicates a new file; if the file already exists, an error occurs. Once the file is created, its status changes to 'OLD'.
'SCRATCH'	Indicates a new file that is unnamed (called a scratch file). When the file is closed or the program terminates, the scratch file is deleted.
'REPLACE'	Indicates the file replaces another. If the file to be replaced exists, it is deleted and a new file is created with the same name.

	If the file to be replaced does not exist, a new file is created and its status changes to 'OLD'.
'UNKNOWN'	Indicates the file may or may not exist. If the file does not exist, a new file is created and its status changes to 'OLD'.

Scratch files go into a temporary directory and are visible while they are open. Scratch files are deleted when the unit is closed or when the program terminates normally, whichever occurs first.

To specify the path for scratch files, you can use one of the following environment variables:

- On Windows* OS: FORT_TMPDIR, TMP, or TEMP, searched in that order
- On Linux* OS and Mac OS X: FORT_TMPDIR or TMPDIR, searched in that order

If no environment variable is defined, the default is the current directory.

The default is 'UNKNOWN'. This is also the default if you implicitly open a file by using WRITE. However, if you implicitly open a file using READ, the default is 'OLD'. If you specify compiler option f66 (or OPTIONS/NOF77), the default is 'NEW'.

NOTE. The STATUS specifier can also appear in CLOSE statements to indicate the file's status after it is closed. However, in CLOSE statements the STATUS values are the same as those listed for the DISPOSE specifier.

See Also

- OPEN Statement Overview
- f66 compiler option

OPEN: TITLE Specifier (W*32, W*64)

The TITLE specifier indicates the name of a child window in a QuickWin application. It takes the following form:

TITLE = *name*

name Is a character expression.

If TITLE is specified in a non-Quickwin application, a run-time error occurs.

See Also

- OPEN Statement Overview

Building Applications: Using QuickWin Overview

OPEN: TYPE Specifier

TYPE is a nonstandard synonym for STATUS.

OPEN: USEROPEN Specifier

The USEROPEN specifier lets you pass control to a routine that directly opens a file. The file can use system calls or library routines to establish a special context that changes the effect of subsequent Fortran I/O statements.

The USEROPEN specifier takes the following form:

USEROPEN = *function-name*

function-name	Is the name of an external function; it must be of type INTEGER(4) (INTEGER*4). The external function can be written in Fortran, C, or other languages.

If the function is written in Fortran, do not execute a Fortran OPEN statement to open the file named in USEROPEN.

The Intel® Fortran Run-time Library (RTL) I/O support routines call the function named in USEROPEN in place of the system calls normally used when the file is first opened for I/O.

On Windows* systems, the Fortran RTL normally calls CreateFile( ) to open a file. When USEROPEN is specified, the called function opens the file (or pipe, etc.) by using CreateFile( ) and returns the *handle* of the file (return value from CreateFile( )) when it returns control to the calling Fortran program.

On Linux* and Mac OS* X systems, the Fortran RTL normally calls the open function to open a file. When USEROPEN is specified, the called function opens the file by calling open and returns the file descriptor of the file when it returns control to the calling Fortran program.

When opening the file, the called function usually specifies options different from those provided by a normal Fortran OPEN statement.

Examples

The following shows an example on Linux and Mac OS X systems and an example on Windows systems.

Example on Linux and Mac OS X systems:

```
      PROGRAM UserOpenMain
      IMPLICIT NONE
      EXTERNAL      UOPEN
      INTEGER(4)    UOPEN
      CHARACTER(10) :: FileName="UOPEN.DAT"
      INTEGER       :: IOS
      CHARACTER(255):: InqFullName
      CHARACTER(100):: InqFileName
      INTEGER       :: InqLun
      CHARACTER(30) :: WriteOutBuffer="Write_One_Record_to_the_File. "
      CHARACTER(30) :: ReadInBuffer  ="??????????????????????????????"
110   FORMAT( X,"FortranMain: ",A," Created (iostat=",I0,")")
115   FORMAT( X,"FortranMain: ",A,": Creation Failed (iostat=",I0,")")
120   FORMAT( X,"FortranMain: ",A,": ERROR: INQUIRE Returned Wrong FileName")
130   FORMAT( X,"FortranMain: ",A,": ERROR: ReadIn and WriteOut Buffers Do Not Match")
      WRITE(*,'(X,"FortranMain: Test the USEROPEN Facility of Open")')
      OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='REPLACE',USEROPEN=UOPEN, &
           IOSTAT=ios, ACTION='READWRITE')

!     When the OPEN statement is executed, the uopen_ function receives control.
!     The uopen_ function opens the file by calling open(), and subsequently
!     returns control with the handle returned by open().
      IF (IOS .EQ. 0)  THEN
         WRITE(*,110) TRIM(FileName), IOS
         INQUIRE(10, NAME=InqFullName)
         CALL ParseForFileName(InqFullName,InqFileName)
         IF (InqFileName .NE. FileName) THEN
```

```
            WRITE(*,120) TRIM(FileName)
         END IF
      ELSE
         WRITE(*,115) TRIM(FileName), IOS
         GOTO 9999
      END IF
      WRITE(10,*) WriteOutBuffer
      REWIND(10)
      READ(10,*) ReadInBuffer
      IF (ReadinBuffer .NE. WriteOutbuffer) THEN
         WRITE(*,130) TRIM(FileName)
      END IF
      CLOSE(10)
      WRITE(*,'(X,"FortranMain: Test of USEROPEN Completed")')
9999  CONTINUE
      END
!----------------------------------------------------------------
! SUBROUTINE: ParseForFileName
!             Takes a full pathname and returns the filename
!             with its extension.
```

```
!---------------------------------------------------------------
      SUBROUTINE ParseForFileName(FullName,FileName)
      CHARACTER(255):: FullName
      CHARACTER(255):: FileName
      INTEGER       :: P
      P = INDEX(FullName,'/',.TRUE.)
      FileName = FullName(P+1:)
      END
//
// File: UserOpen_Sub.c
//
// This routine opens a file using data passed from the Intel(c) Fortran OPEN statement.
//
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <errno.h>
int uopen_ ( char *file_name,   /* access read: name of the file to open (null terminated) */
             int  *open_flags,  /* access read: READ/WRITE, see file.h or open(2) */
             int  *create_mode, /* access read: set if the file is to be created */
             int  *unit_num,    /* access read: logical unit number to be opened */
             int  filenam_len ) /* access read: number of characters in file_name */
{
    /*
    ** The returned value is the following:
```

```
    **    value >= 0 is a valid file descriptor
    **    value < 0 is an error
    */
    int return_value;
    printf(" %s: Opening FILENAME = %s\n", __FILE__, file_name);
    printf(" %s: open_flags = 0x%8.8x\n", __FILE__, *open_flags);
    if ( *open_flags & O_CREAT ) {
       printf(" %s: the file is being created, create_mode = 0x%8.8x\n", __FILE__, *create_mode);
    }
    printf(" %s: open() ", __FILE__);
    return_value = open(file_name, *open_flags, *create_mode);
    if (return_value < 0) {
        printf("FAILED.\n");
    } else {
        printf("SUCCEEDED.\n");
    }
    return (return_value);
} /* end of uopen_() */
```

Example on Windows systems:

In the calling Fortran program, the function named in USEROPEN must first be declared in an EXTERNAL statement. For example, the following Fortran code might be used to call the USEROPEN procedure UOPEN:

```
  IMPLICIT INTEGER (A-Z)
  EXTERNAL UOPEN
  ...
  OPEN(UNIT=10,FILE='UOPEN.DAT',STATUS='NEW',USEROPEN=UOPEN)
```

When the OPEN statement is executed, the UOPEN function receives control. The function opens the file by calling CreateFile( ), performs whatever operations were specified, and subsequently returns control (with the *handle* returned by CreateFile( )) to the calling Fortran program.

Here is what the UOPEN function might look like:

```
        INTEGER FUNCTION UOPEN( FILENAME,      &
                                DESIRED_ACCESS, &
                                SHARE_MODE,     &
                                A_NULL,         &
                                CREATE_DISP,    &
                                FLAGS_ATTR,     &
                                B_NULL,         &
                                UNIT,           &
                                FLEN )
        !DEC$ ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
        !DEC$ ATTRIBUTES REFERENCE :: FILENAME
        !DEC$ ATTRIBUTES REFERENCE :: DESIRED_ACCESS
        !DEC$ ATTRIBUTES REFERENCE :: SHARE_MODE
        !DEC$ ATTRIBUTES REFERENCE :: CREATE_DISP
        !DEC$ ATTRIBUTES REFERENCE :: FLAGS_ATTR
        !DEC$ ATTRIBUTES REFERENCE :: UNIT
        USE IFWIN
        IMPLICIT INTEGER (A-Z)
        CHARACTER*(FLEN) FILENAME
        TYPE(T_SECURITY_ATTRIBUTES), POINTER :: NULL_SEC_ATTR
! Set the FILE_FLAG_WRITE_THROUGH bit in the flag attributes to CreateFile( )
! (for whatever reason)
        FLAGS_ATTR = FLAGS_ATTR + FILE_FLAG_WRITE_THROUGH
! Do the CreateFile( ) call and return the status to the Fortran rtl
        STS = CreateFile( FILENAME,             &
                          DESIRED_ACCESS,       &
                          SHARE_MODE,           &
                          NULL_SEC_ATTR,        &
```

```
                    CREATE_DISP,          &
                    FLAGS_ATTR,           &
                    0 )
      UOPEN = STS
      RETURN
      END
```

The UOPEN function is declared to use the cdecl calling convention, so it matches the Fortran rtl declaration of a useropen routine.

The following function definition and arguments are passed from the Intel Fortran Run-time Library to the function named in USEROPEN:

```
      INTEGER FUNCTION UOPEN( FILENAME,       &
                              DESIRED_ACCESS, &
                              SHARE_MODE,     &
                              A_NULL,         &
                              CREATE_DISP,    &
                              FLAGS_ATTR,     &
                              B_NULL,         &
                              UNIT,           &
                              FLEN )
      !DEC$ ATTRIBUTES C, ALIAS:'_UOPEN' :: UOPEN
      !DEC$ ATTRIBUTES REFERENCE :: DESIRED_ACCESS
      !DEC$ ATTRIBUTES REFERENCE :: SHARE_MODE
      !DEC$ ATTRIBUTES REFERENCE :: CREATE_DISP
      !DEC$ ATTRIBUTES REFERENCE :: FLAGS_ATTR
      !DEC$ ATTRIBUTES REFERENCE :: UNIT
```

The first 7 arguments correspond to the CreateFile( ) api arguments. The value of these arguments is set according the caller's OPEN( ) arguments:

FILENAME	Is the address of a null terminated character string that is the name of the file.
DESIRED_ACCESS	Is the desired access (read-write) mode passed by reference.

SHARE_MODE	Is the file sharing mode passed by reference.
A_NULL	Is always null. The Fortran runtime library always passes a NULL for the pointer to a SECURITY_ATTRIBUTES structure in its CreateFile( ) call.
CREATE_DISP	Is the creation disposition specifying what action to take on files that exist, and what action to take on files that do not exist. It is passed by reference.
FLAGS_ATTR	Specifies the file attributes and flags for the file. It is passed by reference.
B_NULL	Is always null. The Fortran runtime library always passes a NULL for the handle to a template file in it's CreateFile( ) call.

The last 2 arguments are the Fortran unit number and length of the file name:

UNIT	Is the Fortran unit number on which this OPEN is being done. It is passed by reference.
FLEN	Is the length of the file name, not counting the terminating null, and passed by value.

REWIND Statement Overview

The REWIND statement positions a sequential or direct access file at the beginning of the file (the initial point). For more information, see REWIND.

WAIT Statement Overview

The WAIT statement performs a wait operation for specified pending asynchronous data transfer operations. For more information, see WAIT.

Compilation Control Lines and Statements

54

In addition to specifying options on the compiler command line, you can specify the following lines anjd statements in a program unit to influence compilation:

- The INCLUDE Line

  Incorporates external source code into programs.

- The OPTIONS Statement

  Sets options usually specified in the compiler command line. OPTIONS statement settings override command line options.

Directive Enhanced Compilation 55

Directive enhanced compilation is performed by using compiler directives. Compiler directives are special commands that let you perform various tasks during compilation. They are similar to compiler options, but can provide more control within your program.

Compiler directives are preceded by a special prefix that identifies them to the compiler.

Syntax Rules for Compiler Directives

The following syntax rules apply to all general and OpenMP* Fortran compiler directives. You must follow these rules precisely to compile your program properly and obtain meaningful results.

A directive prefix (tag) takes one of the following forms:

General compiler directives:	*c*DEC$
OpenMP Fortran compiler directives:	*c*$OMP
c	Is one of the following: C (or c), !, or *.

The following prefix forms can be used in place of *c*DEC$: *c*DIR$ or !MS$.

The following are source form rules for directive prefixes:

- In fixed and tab source forms, prefixes begin with C (or c), *, or !.

  The prefix must appear in columns 1 through 5; column 6 must be a blank or a tab[3]. From column 7 on, blanks are insignificant, so the directive can be positioned anywhere on the line after column 6.
- In free source form, prefixes begin with !.

  The prefix can appear in any column, but it cannot be preceded by any nonblank, nontab characters on the same line.
- In all source forms, directives spelled with two keywords can be separated by an optional space; for example, "LOOP COUNT" and "LOOPCOUNT" are both valid spellings for the same directive. However, when a directive name is preceded by the prefix "NO", this is not considered to be two keywords. For example, "NO PREFETCH" is not a valid spelling; the only valid spelling for this directive is "NOPREFETCH".

A compiler directive ends in column 72 (or column 132, if compiler option extend-source is specified).

[3] Except for prefix !MS$

General compiler directives cannot be continued. OpenMP Fortran directives can be continued.

A comment beginning with an ! can follow a compiler directive on the same line.

Additional Fortran statements (or directives) cannot appear on the same line as the compiler directive.

Compiler directives cannot appear within a continued Fortran statement.

Blank common used in a compiler directive is specified by two slashes (/ /).

If the source line starts with a valid directive prefix but the directive is not recognized, the compiler displays an informational message and ignores the line.

General Compiler Directives

Intel® Fortran provides several general-purpose compiler directives to perform tasks during compilation. You do not need to specify a compiler option to enable general directives.

The following general compiler directives are available:

- ALIAS

  Specifies an alternate external name to be used when referring to external subprograms.
- ASSUME_ALIGNED

  Specifies that an entity in memory is aligned.
- ATTRIBUTES

  Specifies properties for data objects and procedures.
- DECLARE and NODECLARE

  Generates or disables warnings for variables that have been used but not declared.
- DEFINE and UNDEFINE

  Defines (or undefines) a symbolic variable whose existence (or value) can be tested during conditional compilation.
- DISTRIBUTE POINT

  Specifies distribution for a DO loop.
- FIXEDFORMLINESIZE

  Sets the line length for fixed-form source code.
- FREEFORM and NOFREEFORM

  Specifies free-format or fixed-format source code.
- IDENT

Specifies an identifier for an object module.

- IF and IF DEFINED

  Specifies a conditional compilation construct.
- INTEGER

  Specifies the default integer kind.
- IVDEP

  Assists the compiler's dependence analysis of iterative DO loops.
- LOOP COUNT

  Specifies the loop count for a DO loop; this assists the optimizer.
- MEMORYTOUCH

  Ensures that a specific memory location is updated dynamically.
- MEMREF_CONTROL

  Lets you provide cache hints on prefetches, loads, and stores.
- MESSAGE

  Specifies a character string to be sent to the standard output device during the first compiler pass.
- OBJCOMMENT

  Specifies a library search path in an object file.
- OPTIMIZE and NOOPTIMIZE

  Enables or disables optimizations.
- OPTIONS

  Affects data alignment and warnings about data alignment.
- PACK

  Specifies the memory starting addresses of derived-type items.
- PARALLEL and NOPARALLEL

  Facilitates or prevents auto-parallelization for the immediately following DO loop.
- PREFETCH and NOPREFETCH

  Enables or disables a data prefetch from memory.
- PSECT

  Modifies certain characteristics of a common block.
- REAL

Specifies the default real kind.

- STRICT and NOSTRICT

  Disables or enables language features not found in the language standard specified on the command line (Fortran 2003, Fortran 95, or Fortran 90).

- SWP and NOSWP

  Enables or disables software pipelining for a DO loop.

- UNROLL and NOUNROLL

  Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop.

- UNROLL_AND_JAM and NOUNROLL_AND_JAM

  Enables or disables loop unrolling and jamming.

- VECTOR ALIGNED and VECTOR UNALIGNED

  Specifies that all data is aligned or no data is aligned in a DO loop.

- VECTOR ALWAYS and NOVECTOR

  Enables or disables vectorization of a DO loop.

- VECTOR TEMPORAL and VECTOR NONTEMPORAL

  Controls how the 'stores' of register contents to storage are performed (streaming versus non-streaming).

Rules for General Directives that Affect DO Loops

This table lists the general directives that affect DO loops:

DISTRIBUTE POINT	NOUNROLL_AND_JAM	VECTOR ALIGNED
IVDEP	NOVECTOR[1]	VECTOR ALWAYS
LOOP COUNT	PARALLEL	VECTOR NONTEMPORAL[1]
NOPARALLEL	PREFETCH	VECTOR NOVECTOR
NOPREFETCH	SWP[2]	VECTOR TEMPORAL[1]
NOSWP[2]	UNROLL	VECTOR UNALIGNED
NOUNROLL	UNROLL_AND_JAM	

[1] i32, i64em

[2] i64 only

The following rules apply to all of these directives:

- The directive must precede the DO statement for each DO loop it affects.
- No source code lines, other than the following, can be placed between the directive statement and the DO statement:
  - One of the other general directives that affect DO loops
  - An OpenMP* Fortran PARALLEL DO directive
  - Comment lines
  - Blank lines

Other rules may apply to these directives. For more information, see the description of each directive.

Rules for Loop Directives that Affect Array Assignment Statements

When certain loop directives precede an array assignment statement, they affect the implicit loops that are generated by the compiler.

The following loop directives can affect array assignment statements:

IVDEP	NOVECTOR[1]	VECTOR ALIGNED
LOOP COUNT	PARALLEL	VECTOR ALWAYS
NOPARALLEL	PREFETCH	VECTOR NONTEMPORAL[1]
NOPREFETCH	SWP[2]	VECTOR NOVECTOR[1]
NOSWP[2]	UNROLL	VECTOR TEMPORAL[1]
NOUNROLL	UNROLL_AND_JAM	VECTOR UNALIGNED

[1] i32, i64em

[2] i64 only

Only one of the above directives can precede the array assignment statement (`one-dimensional-array = expression`) to affect it.

Other rules may apply to these directives. For more information, see the description of each directive.

Examples

Consider the following:

```
      REAL A(10), B(10)
      ...
!DEC$ IVDEP
      A = B + 3
```

This has the same effect as writing the following explicit loop:

```
!DEC$ IVDEP
      DO I = 1, 10
         A (I) = B (I) + 3
      END DO
```

OpenMP* Fortran Compiler Directives

Intel® Fortran provides OpenMP* Fortran compiler directives that comply with OpenMP Fortran Application Program Interface (API) specification Version 1.1 and most of Version 2.0.

To use these directives, you must specify compiler option -openmp (Linux and Mac OS X) or /Qopenmp (Windows).

This section discusses data scope attribute clauses, conditional compilation rules, nesting and binding rules, and the following directives:

- ATOMIC directive

  Specifies that a specific memory location is to be updated dynamically.
- BARRIER directive

  Synchronizes all the threads in a team.
- CRITICAL directive

  Restricts access for a block of code to only one thread at a time.
- DO directive

  Specifies that the iterations of the immediately following DO loop must be executed in parallel.
- FLUSH directive

  Specifies synchronization points where the implementation must have a consistent view of memory.

- MASTER directive

  Specifies a block of code to be executed by the master thread of the team.
- ORDERED directive

  Specifies a block of code to be executed sequentially.
- PARALLEL directive

  Defines a parallel region.
- PARALLEL DO directive

  Defines a parallel region that contains a single DO directive.
- PARALLEL SECTIONS directive

  Defines a parallel region that contains SECTIONS directives.
- PARALLEL WORKSHARE directive

  Defines a parallel region that contains a single WORKSHARE directive.
- SECTIONS directive

  Specifies a block of code to be divided among threads in a team (a worksharing area).
- SINGLE directive

  Specifies a block of code to be executed by only one thread in a team.
- TASK directive

  Defines a task region.
- TASKWAIT directive

  Specifies a wait on the completion of child tasks generated since the beginning of the current task.
- THREADPRIVATE directive

  Makes named common blocks private to a thread but global within the thread.
- WORKSHARE directive

  Divides the work of executing a block of statements or constructs into separate units.

The OpenMP parallel directives can be grouped into the categories shown in the following table:

Table 677: Categories of OpenMP Fortran Parallel Directives

Category	Description
Parallel region	Defines a parallel region: PARALLEL
Task region	Defines a task region: TASK

Category	Description
Work-sharing	Divide the execution of the enclosed block of code among the members of the team that encounter it: DO and SECTIONS
Combined parallel work-sharing	Shortcut for denoting a parallel region that contains only one work-sharing construct: PARALLEL DO and PARALLEL SECTIONS
Synchronization	Provide various aspects of synchronization; for example, access to a block of code, or execution order of statements within a block of code: ATOMIC, BARRIER, CRITICAL, FLUSH, MASTER, ORDERED, and TASKWAIT.
Data Environment	Control the data environment during the execution of parallel constructs: THREADPRIVATE

Note that certain general directives and rules can affect DO loops. For more information, see Rules for General Directives that Affect DO Loops.

Data Scope Attribute Clauses

Some of the OpenMP* Fortran directives have clauses (or options) you can specify to control the scope attributes of variables for the duration of the directive.

Other clauses (or options) are available for some OpenMP Fortran directives. For more information, see each directive description.

See Also

- OpenMP* Fortran Compiler Directives
- COPYIN
- COPYPRIVATE
- DEFAULT
- FIRSTPRIVATE
- LASTPRIVATE
- PRIVATE
- REDUCTION
- SHARED

Conditional Compilation Rules

The OpenMP* Fortran API lets you conditionally compile Intel® Fortran statements if you use the appropriate directive prefix.

The prefix depends on which source form you are using, although !$ is valid in all forms.

The prefix must be followed by a valid Intel Fortran statement on the same line.

Free Source Form

The free source form conditional compilation prefix is !$. This prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Free-form source rules apply to the directive line.

Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix with optional white space before and after the ampersand.

Fixed Source Form

For fixed source form programs, the conditional compilation prefix is one of the following: !$, C$ (or c$), or *$.

The prefix must start in column one and appear as a single string with no intervening white space. Fixed-form source rules apply to the directive line.

Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six. For example, the following forms for specifying conditional compilation are equivalent:

```
c23456789
!$    IAM = OMP_GET_THREAD_NUM( ) +
!$   *      INDEX
#IFDEF _OPENMP
      IAM = OMP_GET_THREAD_NUM( ) +
     *      INDEX
#ENDIF
```

Nesting and Binding Rules

This section describes the dynamic nesting and binding rules for OpenMP* Fortran API directives.

Binding Rules

The following rules apply to dynamic binding:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL directive.

Nesting Rules

The following rules apply to dynamic nesting:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Examples

The following example shows nested PARALLEL regions:

```
  c$OMP PARALLEL DEFAULT(SHARED)
  c$OMP DO
              DO I =1, N
  c$OMP PARALLEL SHARED(I,N)
  c$OMP DO
                  DO J =1, N
                      CALL WORK(I,J)
          END DO
  c$OMP END   PARALLEL
              END DO
  c$OMP END PARALLEL
```

Note that the inner and outer DO directives bind to different PARALLEL regions.

The following example shows a variation of the preceding example:

```
c$OMP PARALLEL DEFAULT(SHARED)
c$OMP DO
                DO I =1, N
                    CALL SOME_WORK(I,N)
                END DO
c$OMP END PARALLEL
      ...
      SUBROUTINE SOME_WORK(I,N)
c$OMP PARALLEL DEFAULT(SHARED)
c$OMP DO
                DO J =1, N
                    CALL WORK(I,J)
        END DO
c$OMP END PARALLEL
            RETURN
            END
```

Scope and Association

56

Program entities are identified by names, labels, input/output unit numbers, operator symbols, or assignment symbols. For example, a variable, a derived type, or a subroutine is identified by its name.

Scope refers to the area in which a name is recognized. A scoping unit is the program or part of a program in which a name is defined or known. It can be any of the following:

- An entire executable program
- A single scoping unit
- A single statement (or part of a statement)

The region of the program in which a name is known and accessible is referred to as the scope of that name. These different scopes allow the same name to be used for different things in different regions of the program.

Association is the language concept that allows different names to refer to the same entity in a particular region of a program.

Scope

Program entities have the following kinds of scope (as shown in the table below):

- Global

  Entities that are accessible throughout an executable program. The name of a global entity must be unique. It cannot be used to identify any other global entity in the same executable program.

- Scoping unit (Local scope)

  Entities that are declared within a scoping unit. These entities are local to that scoping unit. The names of local entities are divided into classes (see the table below).

  A *scoping unit* is one of the following:

  - A derived-type definition
  - A procedure interface body (excluding any derived-type definitions and interface bodies contained within it)
  - A program unit or subprogram (excluding any derived-type definitions, interface bodies, and subprograms contained within it)

  A scoping unit that immediately surrounds another scoping unit is called the host scoping unit. Named entities within the host scoping unit are accessible to the nested scoping unit by host association. (For information about host association, see Use and Host Association.)

Once an entity is declared in a scoping unit, its name can be used throughout that scoping unit. An entity declared in another scoping unit is a different entity even if it has the same name and properties.

Within a scoping unit, a local entity name that is not generic must be unique within its class. However, the name of a local entity in one class can be used to identify a local entity of another class.

Within a scoping unit, a generic name can be the same as any one of the procedure names in the interface block.

A component name has the same scope as the derived type of which it is a component. It can appear only within a component designator of a structure of that type.

For information on interactions between local and global names, see the table below.

- Statement

  Entities that are accessible only within a statement or part of a statement; such entities cannot be referenced in subsequent statements.

  The name of a statement entity can also be the name of a global or local entity in the same scoping unit; in this case, the name is interpreted within the statement as that of the statement entity.

Table 678: Scope of Program Entities

Entity	Scope	
Program units	Global	
Common blocks[1]	Global	
External procedures	Global	
Intrinsic procedures	Global[2]	
Module procedures	Local	Class I
Internal procedures	Local	Class I
Dummy procedures	Local	Class I
Statement functions	Local	Class I
Derived types	Local	Class I
Components of derived types	Local	Class II

Entity	Scope	
Named constants	Local	Class I
Named constructs	Local	Class I
Namelist group names	Local	Class I
Generic identifiers	Local	Class I
Argument keywords in procedures	Local	Class III
Variables that can be referenced throughout a subprogram	Local	Class I
Variables that are dummy arguments in statement functions	Statement	
DO variables in an implied-DO list[3] of a DATA or FORALL statement, or an array constructor	Statement	
Intrinsic operators	Global	
Defined operators	Local	
Statement labels	Local	
External I/O unit numbers	Global	
Intrinsic assignment	Global[4]	
Defined assignment	Local	

[1] Names of common blocks can also be used to identify local entities.

[2] If an intrinsic procedure is not used in a scoping unit, its name can be used as a local entity within that scoping unit. For example, if intrinsic function COS is not used in a program unit, COS can be used as a local variable there.

[3] The DO variable in an implied-DO list of an I/O list has local scope.

Entity	Scope
[4] The scope of the assignment symbol (=) is global, but it can identify additional operations (see Defining Generic Assignment).	

Scoping units can contain other scoping units. For example, the following shows six scoping units:

```
MODULE MOD_1                    ! Scoping unit 1
   ...                          ! Scoping unit 1
CONTAINS                        ! Scoping unit 1
  FUNCTION FIRST                ! Scoping unit 2
    TYPE NAME                   ! Scoping unit 3
    ...                         ! Scoping unit 3
    END TYPE NAME               ! Scoping unit 3
    ...                         ! Scoping unit 2
  CONTAINS                      ! Scoping unit 2
    SUBROUTINE SUB_B            ! Scoping unit 4
      TYPE PROCESS      ! Scoping unit 5
      ...                       ! Scoping unit 5
      END TYPE PROCESS  ! Scoping unit 5
      INTERFACE                 ! Scoping unit 5
        SUBROUTINE SUB_A        ! Scoping unit 6
        ...                     ! Scoping unit 6
        END SUBROUTINE SUB_A    ! Scoping unit 6
      END INTERFACE             ! Scoping unit 5
    END SUBROUTINE SUB_B        ! Scoping unit 4
  END FUNCTION   FIRST          ! Scoping unit 2
END MODULE                      ! Scoping unit 1
```

See Also

- Scope and Association
- Derived data types

- Defining Generic Names for Procedures
- Intrinsic procedures
- Program Units and Procedures
- Use and host association
- Defining Generic Operators
- Defining Generic Assignment
- PRIVATE and PUBLIC Attributes and Statements

Unambiguous Generic Procedure References

When a generic procedure reference is made, a specific procedure is invoked. If the following rules are used, the generic reference will be unambiguous:

- Within a scoping unit, two procedures that have the same generic name must both be subroutines (or both be functions). One of the procedures must have a nonoptional dummy argument that is one of the following:
  - Not present by position or argument keyword in the other argument list
  - Is present, but has different type and kind parameters, or rank
- Within a scoping unit, two procedures that have the same generic operator must both have the same number of arguments or both define assignment. One of the procedures must have a dummy argument that corresponds by position in the argument list to a dummy argument of the other procedure that has a different type and kind parameters, or rank.

When an interface block extends an intrinsic procedure, operator, or assignment, the rules apply as if the intrinsic consists of a collection of specific procedures, one for each allowed set of arguments.

When a generic procedure is accessed from a module, the rules apply to all the specific versions, even if some of them are inaccessible by their specific names.

See Also

- Scope and Association
- Defining Generic Names for Procedures

Resolving Procedure References

The procedure name in a procedure reference is either established to be generic or specific, or is not established. The rules for resolving a procedure reference differ depending on whether the procedure is established and how it is established.

This section discusses the following topics:

- References to Generic Names
- References to Specific Names
- References to Nonestablished Names

References to Generic Names

Within a scoping unit, a procedure name is established to be generic if any of the following is true:

- The scoping unit contains an interface block with that procedure name.
- The procedure name matches the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is established to be generic in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be generic in a host scoping unit.

To resolve a reference to a procedure name established to be generic, the following rules are used in the order shown:

1. If an interface block with that procedure name appears in one of the following, the reference is to the specific procedure providing that interface:

   a. The scoping unit that contains the reference

   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with one of the specific interfaces of the interface block.

2. If the procedure name is specified with the INTRINSIC attribute in one of the following, the reference is to that intrinsic procedure:

   a. The same scoping unit

   b. A module made accessible by a USE statement in the scoping unit

   The reference must be consistent with the interface of that intrinsic procedure.

3. If the following is true, the reference is resolved by applying rules 1 and 2 to the host scoping unit:

   a. The procedure name is established to be generic in the host scoping unit

   b. There is agreement between the scoping unit and the host scoping unit as to whether the procedure is a function or subroutine name.

4. If none of the preceding rules apply, the reference must be to the generic intrinsic procedure with that name. The reference must be consistent with the interface of that intrinsic procedure.

Examples

The following example shows how a module can define three separate procedures, and a main program give them a generic name DUP through an interface block. Although the main program calls all three by the generic name, there is no ambiguity since the arguments are of different data types, and DUP is a function rather than a subroutine. The module UN_MOD must give each procedure a different name.

```
MODULE UN_MOD
!
CONTAINS
   subroutine dup1(x,y)
   real x,y
   print *, ' Real arguments', x, y
   end subroutine dup1
   subroutine dup2(m,n)
   integer m,n
   print *, ' Integer arguments', m, n
   end subroutine dup2
   character function dup3 (z)
   character(len=2) z
   dup3 = 'String argument '// z
   end function dup3
END MODULE
program unclear
!
! shows how to use generic procedure references
USE UN_MOD
INTERFACE DUP
   MODULE PROCEDURE dup1, dup2, dup3
END INTERFACE
```

```
real a,b
integer c,d
character (len=2) state
a = 1.5
b = 2.32
c = 5
d = 47
state = 'WA'
call dup(a,b)
call dup(c,d)
print *, dup(state)       !actual output is 'S' only
END
```

Note that the function DUP3 only prints one character, since module UN_MOD specifies no length parameter for the function result.

If the dummy arguments x and y for DUP were declared as integers instead of reals, then any calls to DUP would be ambiguous. If this is the case, a compile-time error results.

The subroutine definitions, DUP1, DUP2, and DUP3, must have different names. The generic name is specified in the first line of the interface block, and in the example is DUP.

References to Specific Names

In a scoping unit, a procedure name is established to be specific if it is not established to be generic and any of the following is true:

- The scoping unit contains an interface body with that procedure name.
- The scoping unit contains an internal procedure, module procedure, or statement function with that procedure name.
- The procedure name is the same as the name of a generic intrinsic procedure, and it is specified with the INTRINSIC attribute in that scoping unit.
- The procedure name is specified with the EXTERNAL attribute in that scoping unit.
- The procedure name is established to be specific in a module, and the scoping unit contains a USE statement making that procedure name accessible.
- The scoping unit contains no declarations for that procedure name, but the procedure name is established to be specific in a host scoping unit.

To resolve a reference to a procedure name established to be specific, the following rules are used in the order shown:

1. If either of the following is true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram, and it contains an interface body with that procedure name.

   b. The procedure name has been declared EXTERNAL, and the procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.
2. If the scoping unit contains an interface body or the procedure name has been declared EXTERNAL, and Rule 1 does not apply, the reference is to an external procedure with that name.
3. If the scoping unit contains an internal procedure or statement function with that procedure name, the reference is to that entity.
4. If the procedure name has been declared INTRINSIC in the scoping unit, the reference is to the intrinsic procedure with that name.
5. If the scoping unit contains a USE statement that makes the name of a module procedure accessible, the reference is to that procedure. (The USE statement allows renaming, so the name referenced may differ from the name of the module procedure.)
6. If none of the preceding rules apply, the reference is resolved by applying these rules to the host scoping unit.

References to Nonestablished Names

In a scoping unit, a procedure name is not established if it is not determined to be generic or specific.

To resolve a reference to a procedure name that is not established, the following rules are used in the order shown:

1. If both of the following are true, the dummy argument is a dummy procedure and the reference is to that dummy procedure:

   a. The scoping unit is a subprogram.

   b. The procedure name is a dummy argument of that subprogram.

   The procedure invoked by the reference is the one supplied as the corresponding actual argument.

2. If both of the following are true, the procedure is an intrinsic procedure and the reference is to that intrinsic procedure:
   a. The procedure name matches the name of an intrinsic procedure.
   b. There is agreement between the intrinsic procedure definition and the reference of the name as a function or subroutine.
3. If neither of the preceding rules apply, the reference is to an external procedure with that name.

See Also

- Resolving Procedure References
- Function references
- USE statement
- CALL Statement
- Defining Generic Names for Procedures

Association

Association allows different program units to access the same value through different names. Entities are associated when each is associated with the same storage location.

There are three kinds of association:

- Name association
- Pointer association
- Storage association

The following example shows name, pointer, and storage association between an external program unit and an external procedure.

Example of Name, Pointer, and Storage Association

```
! Scoping Unit 1: An external program unit

REAL A, B(4)

REAL, POINTER :: M(:)

REAL, TARGET :: N(12)

COMMON /COM/...

EQUIVALENCE (A, B(1))         ! Storage association between A and B(1)

M => N                        ! Pointer association

CALL P (actual-arg,...)

...


! Scoping Unit 2: An external procedure

SUBROUTINE P (dummy-arg,...)  ! Name and storage association between
                              !    these arguments and the calling
                              !    routine's arguments in scoping unit 1

  COMMON /COM/...             ! Storage association with common block COM

                              !    in scoping unit 1

  REAL Y

  CALL Q (actual-arg,...)

  CONTAINS

    SUBROUTINE Q (dummy-arg,...) ! Name and storage association between

                                 !   these arguments and the calling

                                 !   routine's arguments in host procedure
                                 !   P (subprogram Q has host association
                                 !   with procedure P)

      Y = 2.0*(Y-1.0)          ! Name association with Y in host procedure P

    ...
```

Name Association

Name association allows an entity to be accessed from different scoping units by the same name or by different names. There are three types of name association: argument, use, and host.

Argument Association

Arguments are the values passed to and from functions and subroutines through calling program argument lists.

Execution of a procedure reference establishes argument association between an actual argument and its corresponding dummy argument. The name of a dummy argument can be different from the name of its associated actual argument (if any).

When the procedure completes execution, the argument association is terminated.

See Also

- Name Association
- Argument Association

Use and Host Association

Use association allows the entities in a module to be accessible to other scoping units. The mechanism for use association is the USE statement. The USE statement provides access to all public entities in the module, unless ONLY is specified. In this case, only the entities named in the ONLY list can be accessed.

Host association allows the entities in a host scoping unit to be accessible to an internal procedure, derived-type definition, or module procedure contained within the host. The accessed entities are known by the same name and have the same attributes as in the host. Entities that are local to a procedure are not accessible to its host.

Use or host association remains in effect throughout the execution of the executable program.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible. A name that appears in the scoping unit as an external name in an EXTERNAL statement is a global name, and any entity of the host that has this as its nongeneric name is inaccessible.

An interface body does not access named entities by host association, but it can access entities by use association.

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association can be changed within the procedure. After execution of the procedure, the pointer association remains current, unless the execution caused the target to become undefined. If this occurs, the host associated pointer becomes undefined.

NOTE. Implicit declarations can cause problems for host association. It is recommended that you use IMPLICIT NONE in both the host and the contained procedure, and that you explicitly declare all entities.

When all entities are explicitly declared, local declarations override host declarations, and host declarations that are not overridden are available in the contained procedure.

Examples

The following example shows host and use association:

```
MODULE SHARE_DATA
  REAL Y, Z
END MODULE
PROGRAM DEMO
  USE SHARE_DATA          ! All entities in SHARE_DATA are available
  REAL B, Q               !   through use association.
  ...
  CALL CONS (Y)
CONTAINS
  SUBROUTINE CONS (Y)     ! Y is a local entity (dummy argument).
    REAL C, Y
    ...
    Y = B + C + Q + Z     ! B and Q are available through host association.
    ...                   !   C is a local entity, explicitly declared.  Z
  END SUBROUTINE CONS     !   is available through use association.
END PROGRAM DEMO
```

The following example shows how a host and an internal procedure can use host-associated entities:

```
program INTERNAL
! shows use of internal subroutine and CONTAINS statement
   real a,b,c
   call find
   print *, c
contains
   subroutine find
     read *, a,b
     c = sqrt(a**2 + b**2)
   end subroutine find
end
```

In this example, the variables `a`, `b`, and `c` are available to the internal subroutine `find` through host association. They do not have to be passed as arguments to the internal procedure. In fact, if they are, they become local variables to the subroutine and hide the variables declared in the host program.

Conversely, the host program knows the value of `c`, when it returns from the internal subroutine that has defined `c`.

See Also

- Name Association
- USE statement
- Scope

Pointer Association

A pointer can be associated with a target. At different times during the execution of a program, a pointer can be undefined, associated with different targets, or be disassociated. The initial association status of a pointer is undefined. A pointer can become associated by the following:

- By pointer assignment (pointer => target)

  The target must be associated, or specified with the TARGET attribute. If the target is allocatable, it must be currently allocated.

- By allocation (successful execution of an ALLOCATE statement)

  The ALLOCATE statement must reference the pointer.

A pointer becomes disassociated if any of the following occur:

- The pointer is nullified by a NULLIFY statement.
- The pointer is deallocated by a DEALLOCATE statement.
- The pointer is assigned a disassociated pointer (or the NULL intrinsic function).

When a pointer is associated with a target, the definition status of the pointer is defined or undefined, depending on the definition status of the target. A target is undefined in the following cases:

- If it was never allocated
- If it is not deallocated through the pointer
- If a RETURN or END statement causes it to become undefined

If a pointer is associated with a definable target, the definition status of the pointer can be defined or undefined, according to the rules for a variable.

If the association status of a pointer is disassociated or undefined, the pointer must not be referenced or deallocated.

Whatever its association status, a pointer can always be nullified, allocated, or associated with a target. When a pointer is nullified, it is disassociated. When a pointer is allocated, it becomes associated, but is undefined. When a pointer is associated with a target, its association and definition status are determined by its target.

See Also

- Association
- Pointer assignments
- NULL intrinsic function
- Dynamic Allocation

Storage Association

Storage association is the association of two or more data objects. It occurs when two or more storage sequences share (or are aligned with) one or more *storage units*. Storage sequences are used to describe relationships among variables, common blocks, and result variables.

Storage Units and Storage Sequence

A *storage unit* is a fixed unit of physical memory allocated to certain data. A *storage sequence* is a sequence of storage units. The size of a storage sequence is the number of storage units in the storage sequence. A storage unit can be numeric, character, or unspecified.

A nonpointer scalar of type default real, integer, or logical occupies one numeric storage unit. A nonpointer scalar of type double precision real or default complex occupies two contiguous numeric storage units. In Intel® Fortran, one numeric storage unit corresponds to 4 bytes of memory.

A nonpointer scalar of type default character with character length 1 occupies one character storage unit. A nonpointer scalar of type default character with character length *len* occupies *len* contiguous character storage units. In Intel Fortran, one character storage unit corresponds to 1 byte of memory.

A nonpointer scalar of nondefault data type occupies a single unspecified storage unit. The number of bytes corresponding to the unspecified storage unit differs depending on the data type.

The following table lists the storage requirements (in bytes) for the intrinsic data types:

Table 679: Data Type Storage Requirements

Data Type	Storage Requirements (in bytes)
BYTE	1
LOGICAL	2, 4, or 8 [1]
LOGICAL(1)	1
LOGICAL(2)	2
LOGICAL(4)	4
LOGICAL(8)	8
INTEGER	2, 4, or 8 [1]
INTEGER(1)	1
INTEGER(2)	2
INTEGER(4)	4
INTEGER(8)	8
REAL	4, 8, or 16 [2]
REAL(4)	4
DOUBLE PRECISION	8

Data Type	Storage Requirements (in bytes)
REAL(8)	8
REAL(16)	16
COMPLEX	8, 16, or 32 [2]
COMPLEX(4)	8
DOUBLE COMPLEX	16
COMPLEX(8)	16
COMPLEX(16)	32
CHARACTER	1
CHARACTER*len	len [3]
CHARACTER*(*)	assumed-length [4]

[1] Depending on default integer, LOGICAL and INTEGER can have 2, 4, or 8 bytes. The default allocation is four bytes.

[2] Depending on default real, REAL can have 4, 8, or 16 bytes and COMPLEX can have 8, 16, or 32 bytes. The default allocations are four bytes for REAL and eight bytes for COMPLEX.

[3] The value of len is the number of characters specified. The largest valid value is 2**31-1 on IA-32 architecture; 2**63-1 on Intel® 64 architecture and IA-64 architecture. Negative values are treated as zero.

[4] The assumed-length format *(*) applies to dummy arguments, PARAMETER statements, or character functions, and indicates that the length of the actual argument or function is used. (See Assumed-Length Character Arguments and *Building Applications*.)

A nonpointer scalar of sequence derived type occupies a sequence of storage sequences corresponding to the components of the structure, in the order they occur in the derived-type definition. (A sequence derived type has a SEQUENCE statement.)

A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank.

The definition status and value of a data object affects the definition status and value of any storage-associated entity.

When two objects occupy the same storage sequence, they are totally storage-associated. When two objects occupy parts of the same storage sequence, they are partially associated. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement can cause total or partial storage association of storage sequences.

See Also

- Storage Association
- Assumed-Length Character Arguments
- COMMON
- ENTRY
- EQUIVALENCE

Building Applications for details on the hardware representations of data types

Array Association

A nonpointer array occupies a sequence of contiguous storage sequences, one for each array element, in array element order.

Two or more arrays are associated when each one is associated with the same storage location. They are partially associated when part of the storage associated with one array is the same as part or all of the storage associated with another array.

If arrays with different data types are associated (or partially associated) with the same storage location, and the value of one array is defined (for example, by assignment), the value of the other array becomes undefined. This happens because an element of an array is considered defined only if the storage associated with it contains data of the same type as the array name.

An array element, array section, or whole array is defined by a DATA statement before program execution. (The array properties must be declared in a previous specification statement.) During program execution, array elements and sections are defined by an assignment or input statement, and entire arrays are defined by input statements.

See Also

- Storage Association
- Arrays
- DATA statement
- Array Elements

Deleted and Obsolescent Language Features

57

Fortran 90 identified some FORTRAN 77 features to be obsolescent. Fortran 95 deletes some of these features, and identifies a few more language features to be obsolescent. Features considered obsolescent may be removed from future revisions of the Fortran Standard.

To have these features flagged, you can specify compiler option stand.

NOTE. Intel® Fortran fully supports features deleted from Fortran 95.

Deleted Language Features in Fortran 95

Some language features, considered redundant in FORTRAN 77, are not included in Fortran 95. However, they are still fully supported by Intel® Fortran:

- ASSIGN and assigned GO TO statements
- Assigned FORMAT specifier
- Branching to an END IF statement from outside its IF block
- H edit descriptor
- PAUSE statement
- Real and double precision DO control variables and DO loop control expressions

Intel Fortran flags these features if you specify compiler option stand.

See Also

- Deleted and Obsolescent Language Features
- Obsolescent Language Features in Fortran 90

Obsolescent Language Features in Fortran 95

Some language features, considered redundant in Fortran 90 are identified as obsolescent in Fortran 95.

Intel® Fortran flags these features if you specify compiler option stand.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate returns

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program use a CASE construct to test the value and perform operations.

- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct.

- Assumed-length character functions

  To replace this functionality, it is recommended that you use one of the following:

  - An automatic character-length function, where the length of the function result is declared in a specification expression
  - A subroutine whose arguments correspond to the function result and the function arguments

  Dummy arguments of a function can still have assumed character length; this feature is not obsolescent.

- CHARACTER*(*) form of CHARACTER declaration

  To replace this functionality, it is recommended that you use the Fortran 90 forms of specifying a length selector in CHARACTER declarations (see Declaration Statements for Character Types).

- Computed GO TO statement

  To replace this functionality, it is recommended that you use a CASE construct.

- DATA statements among executable statements

  This functionality has been included since FORTRAN 66, but is considered to be a potential source of errors.

- Fixed source form

  Newer methods of entering data have made this source form obsolescent and error-prone.

  The recommended method for coding is to use free source form.

- Shared DO termination and termination on a statement other than END DO or CONTINUE

  To replace this functionality, it is recommended that you use an END DO statement (see Forms for DO Constructs) or a CONTINUE statement.

- Statement functions

  To replace this functionality, it is recommended that you use an internal function.

Obsolescent Language Features in Fortran 90

Fortran 90 did not delete any of the features in FORTRAN 77, but some FORTRAN 77 features were identified as obsolescent.

Intel® Fortran flags these features if you specify compiler option stand.

Other methods are suggested to achieve the functionality of the following obsolescent features:

- Alternate return (labels in an argument list)

  To replace this functionality, it is recommended that you use an integer variable to return a value to the calling program, and let the calling program test the value and perform operations, using a computed GO TO statement or CASE construct.

- Arithmetic IF

  To replace this functionality, it is recommended that you use an IF statement or construct.

- ASSIGN and assigned GO TO statements

  These statements are usually used to simulate internal procedures, which can now be coded directly.

- Assigned FORMAT specifier (label of a FORMAT statement assigned to an integer variable)

  To replace this functionality, it is recommended that you use character expressions to define format specifications.

- Branching to an END IF statement from outside its IF block

  To replace this functionality, it is recommended that you branch to the statement following the END IF statement (see IF Construct).

- H edit descriptor

  To replace this functionality, it is recommended that you use the character constant edit descriptor (see Character Constant Editing).

- PAUSE statement

  To replace this functionality, it is recommended that you use a READ statement that awaits input data.

- Real and double precision DO control variables and DO loop control expressions

  To replace this functionality, it is recommended that you use integer DO variables and expressions (see DO Constructs).

- Shared DO termination and termination on a statement other than END DO or CONTINUE

  To replace this functionality, it is recommended that you use an END DO statement (see Forms for DO Constructs) or a CONTINUE statement.

Additional Language Features

58

To facilitate compatibility with older versions of Fortran, Intel® Fortran provides the following additional language features:

- The DEFINE FILE statement
- The ENCODE and DECODE statements
- The FIND statement
- The INTERFACE TO statement
- FORTRAN 66 Interpretation of the EXTERNAL Statement
- An alternative syntax for the PARAMETER statement
- The VIRTUAL statement
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for a record specifier
- An alternate syntax for the DELETE statement
- An alternative form for namelist external records
- The integer POINTER statement
- Record structures

These language features are particularly useful in porting older Fortran programs to Fortran 95/90. However, you should avoid using them in new programs on these systems, and in new programs for which portability to other Fortran 95/90 implementations is important.

FORTRAN 66 Interpretation of the EXTERNAL Statement

If you specify compiler option f66, the EXTERNAL statement is interpreted in a way that facilitates compatibility with older versions of Fortran. (The Fortran 95/90 interpretation is incompatible with previous Fortran standards and previous Compaq* implementations.)

The FORTRAN 66 interpretation of the EXTERNAL statement combines the functionality of the INTRINSIC statement with that of the EXTERNAL statement.

This lets you use subprograms as arguments to other subprograms. The subprograms to be used as arguments can be either user-supplied functions or Fortran 95/90 library functions.

The FORTRAN 66 EXTERNAL statement takes the following form:

EXTERNAL [*]*v* [, [*]*v*] ...

*	Specifies that a user-supplied function is to be used instead of a Fortran 95/90 library function having the same name.
v	Is the name of a subprogram or the name of a dummy argument associated with the name of a subprogram.

Description

The FORTRAN 66 EXTERNAL statement declares that each name in its list is an external function name. Such a name can then be used as an actual argument to a subprogram, which then can use the corresponding dummy argument in a function reference or CALL statement.

However, when used as an argument, a complete function reference represents a value, not a subprogram name; for example, SQRT(B) in CALL SUBR(A, SQRT(B), C). It is not, therefore, defined in an EXTERNAL statement (as would be the incomplete reference SQRT).

Examples

The following example shows the FORTRAN 66 EXTERNAL statement:

```
Main Program                          Subprograms
EXTERNAL SIN, COS, *TAN, SINDEG       SUBROUTINE TRIG(X,F,Y)
   .                                  Y = F(X)
   .                                  RETURN
   .                                  END
CALL TRIG(ANGLE, SIN, SINE)
   .
   .                                  FUNCTION TAN(X)
   .                                  TAN = SIN(X)/COS(X)
CALL TRIG(ANGLE, COS, COSINE)         RETURN
   .                                  END
   .
   .
CALL TRIG(ANGLE, TAN, TANGNT)         FUNCTION SINDEG(X)/
   .                                  SINDEG = SIN(X*3.1459/180)
   .                                  RETURN
   .                                  END
CALL TRIG(ANGLED, SINDEG, SINE)
```

The CALL statements pass the name of a function to the subroutine TRIG. The function reference F(X) subsequently invokes the function in the second statement of TRIG. Depending on which CALL statement invoked TRIG, the second statement is equivalent to one of the following:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
Y = SINDEG(X)
```

The functions SIN and COS are examples of trigonometric functions supplied in the Fortran 95/90 library. The function TAN is also supplied in the library, but the asterisk (*) in the EXTERNAL statement specifies that the user-supplied function be used, instead of the library function. The function SINDEG is also a user-supplied function. Because no library function has the same name, no asterisk is required.

See Also

- Additional Language Features
- f66 compiler option

Alternative Syntax for the PARAMETER Statement

The PARAMETER statement discussed here is similar to the one discussed in PARAMETER; they both assign a name to a constant. However, this PARAMETER statement differs from the other one in the following ways:

- Its list is not bounded with parentheses.
- The form of the constant, rather than implicit or explicit typing of the name, determines the data type of the variable.

This PARAMETER statement takes the following form:

PARAMETER *c* = *expr* [, *c* = *expr*] ...

c	Is the name of the constant.
expr	Is an initialization expression. It can be of any data type.

Description

Each name *c* becomes a constant and is defined as the value of expression *expr*. Once a name is defined as a constant, it can appear in any position in which a constant is allowed. The effect is the same as if the constant were written there instead of the name.

The name of a constant cannot appear as part of another constant, except as the real or imaginary part of a complex constant. For example:

```
PARAMETER I=3

PARAMETER M=I.25            ! Not allowed


PARAMETER N=(1.703, I)      ! Allowed
```

The name used in the PARAMETER statement identifies only the name's corresponding constant in that program unit. Such a name can be defined only once in PARAMETER statements within the same program unit.

The name of a constant assumes the data type of its corresponding constant expression. The data type of a parameter constant cannot be specified in a type declaration statement. Nor does the initial letter of the constant's name implicitly affect its data type.

Examples

The following are valid examples of this form of the PARAMETER statement:

```
PARAMETER PI=3.1415927, DPI=3.14159265358979323 8D0

PARAMETER PIOV2=PI/2, DPIOV2=DPI/2

PARAMETER FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS'
```

See Also

- Additional Language Features
- PARAMETER

Alternative Syntax for Binary, Octal, and Hexadecimal Constants

In Intel Fortran, you can use an alternative syntax for binary, octal, and hexadecimal constants. The following table shows the alternative syntax and equivalents:

Constant	Alternative Syntax	Equivalent
Binary	'0..1'B	B'0..1'
Octal	'0..7'O	O'0..7'
Hexadecimal	'0..F'X	Z'0..F'
	X'0..F'	

You can use a quotation mark (") in place of an apostrophe in all the above syntax forms.

For information on the # syntax for integers not in base 10, see Integer Constants.

See Also

- Additional Language Features
- Binary constants
- Octal constants

- Hexadecimal constants

Alternative Syntax for a Record Specifier

In Intel® Fortran, you can specify the following form for a record specifier in an I/O control list:

'*r*

r	Is a numeric expression with a value that represents the position of the record to be accessed using direct access I/O.

The value must be greater than or equal to 1, and less than or equal to the maximum number of records allowed in the file. If necessary, a record number is converted to integer data type before being used.

If this nonkeyword form is used in an I/O control list, it must immediately follow the nonkeyword form of the io-unit specifier.

Alternative Syntax for the DELETE Statement

In Intel® Fortran, you can specify the following form of the DELETE statement when deleting records from a relative file:

DELETE (*io-unit* '*r* [, ERR=*label*] [, IOSTAT=*i-var*])

io-unit	Is the number of the logical unit containing the record to be deleted.
r	Is the positional number of the record to be deleted.
label	Is the label of an executable statement that receives control if an error condition occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

This form deletes the direct access record specified by *r*.

See Also

- Additional Language Features
- DELETE statement

Alternative Form for Namelist External Records

In Intel® Fortran, you can use the following form for an external record:

$*group-name object* = *value* [*object* = *value*] ...$[END]

group-name	Is the name of the group containing the objects to be given values. The name must have been previously defined in a NAMELIST statement in the scoping unit.
object	Is the name (or subobject designator) of an entity defined in the NAMELIST declaration of the group name. The object name must not contain embedded blanks, but it can be preceded or followed by blanks.
value	Is a null value, a constant (or list of constants), a repetition of constants in the form r*c, or a repetition of null values in the form r*.

If more than one *object=value* or more than one value is specified, they must be separated by value separators.

A value separator is any number of blanks, or a comma or slash, preceded or followed by any number of blanks.

See Also

- Additional Language Features
- NAMELIST statement
- Rules for Namelist Sequential READ Statements
- Rules for Namelist Sequential WRITE Statements

Record Structures

The record structure was defined in earlier versions of Intel® Fortran as a language extension. It is still supported, although its functionality has been replaced by standard Fortran 95/90 derived data types. Record structures in existing code can be easily converted to Fortran 95/90 derived type structures for portability, but can also be left in their old form. In most cases, an Intel Fortran record and a Fortran 95/90 derived type can be used interchangeably.

Intel Fortran record structures are similar to Fortran 95/90 derived types.

A *record structure* is an aggregate entity containing one or more elements. (Record elements are also called fields or components.) You can use records when you need to declare and operate on multi-field data structures in your programs.

Creating a record is a two-step process:

1. You must define the form of the record with a multistatement *structure declaration*.
2. You must use a RECORD statement to declare the record as an entity with a name. (More than one RECORD statement can refer to a given structure.)

Examples

Intel Fortran record structures, using only intrinsic types, easily convert to Fortran 95/90 derived types. The conversion can be as simple as replacing the keyword STRUCTURE with TYPE and removing slash (/) marks. The following shows an example conversion:

Record Structure	Fortran 95/90 Derived-Type
STRUCTURE /employee_name/	TYPE employee_name
CHARACTER*25 last_name	CHARACTER*25 last_name
CHARACTER*15 first_name	CHARACTER*15 first_name
END STRUCTURE	END TYPE
STRUCTURE /employee_addr/	TYPE employee_addr
CHARACTER*20 street_name	CHARACTER*20 street_name
INTEGER(2) street_number	INTEGER(2) street_number
INTEGER(2) apt_number	INTEGER(2) apt_number
CHARACTER*20 city	CHARACTER*20 city
CHARACTER*2 state	CHARACTER*2 state
INTEGER(4) zip	INTEGER(4) zip
END STRUCTURE	END TYPE

The record structures can be used as subordinate record variables within another record, such as the `employee_data` record. The equivalent Fortran 90 derived type would use the derived-type objects as components in a similar manner, as shown below:

Record Structure	Fortran 95/90 Derived-Type
STRUCTURE /employee_data/	TYPE employee_data
RECORD /employee_name/ name	TYPE (employee_name) name
RECORD /employee_addr/ addr	TYPE (employee_addr) addr
INTEGER(4) telephone	INTEGER(4) telephone
INTEGER(2) date_of_birth	INTEGER(2) date_of_birth
INTEGER(2) date_of_hire	INTEGER(2) date_of_hire
INTEGER(2) social_security(3)	INTEGER(2) social_security(3)

Record Structure	Fortran 95/90 Derived-Type
LOGICAL(2) married	LOGICAL(2) married
INTEGER(2) dependents	INTEGER(2) dependents
END STRUCTURE	END TYPE

See Also

- Additional Language Features
- Structure Declarations
- References to Record Fields
- Aggregate Assignment
- Structure Declarations
- RECORD Statement
- References to Record Fields
- Aggregate Assignment

Structure Declarations

A *structure declaration* defines the field names, types of data within fields, and order and alignment of fields within a record. Fields and structures can be initialized, but records cannot be initialized. For more information, see STRUCTURE.

Type Declarations within Record Structures

The syntax of a type declaration within a record structure is identical to that of a normal Fortran type statement.

The following rules and behavior apply to type declarations in record structures:

- %FILL can be specified in place of a field name to leave space in a record for purposes such as alignment. This creates an unnamed field.

  %FILL can have an array specification; for example:

  ```
  INTEGER %FILL (2,2)
  ```

  Unnamed fields cannot be initialized. For example, the following statement is invalid and generates an error message:

  ```
  INTEGER %FILL /1980/
  ```

- Initial values can be supplied in field declaration statements. Unnamed fields cannot be initialized; they are always undefined.

- Field names must always be given explicit data types. The IMPLICIT statement does not affect field declarations.
- Any required array dimensions must be specified in the field declaration statements. DIMENSION statements cannot be used to define field names.
- Adjustable or assumed sized arrays and assumed-length CHARACTER declarations are not allowed in field declarations.

Substructure Declarations

A field within a structure can itself be a structured item composed of other fields, other structures, or both. You can declare a substructure in two ways:

- By nesting structure declarations within other structure or union declarations (with the limitation that you cannot refer to a structure inside itself at any level of nesting).

  One or more field names must be defined in the STRUCTURE statement for the substructure, because all fields in a structure must be named. In this case, the substructure is being used as a field within a structure or union.

  Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict.
- By using a RECORD statement that specifies another previously defined record structure, thereby including it in the structure being declared.

See the example in STRUCTURE for a sample structure declaration containing both a nested structure declaration (TIME) and an included structure (DATE).

References to Record Fields

References to record fields must correspond to the kind of field being referenced. Aggregate field references refer to composite structures (and substructures). Scalar field references refer to singular data items, such as variables.

An operation on a record can involve one or more fields.

Record field references take one of the following forms:

Aggregate Field Reference:

record-name [.*aggregate-field-name*] ...

Scalar Field Reference:

record-name [.*aggregate-field-name*] ... .*scalar-field-name*

record-name	Is the name used in a RECORD statement to identify a record.
aggregate-field-name	Is the name of a field that is a substructure (a record or a nested structure declaration) within the record structure identified by the record name.
scalar-field-name	Is the name of a data item (having a data type) defined within a structure declaration.

Description

Records and record fields cannot be used in DATA statements, but individual fields can be initialized in the STRUCTURE definition.

An automatic array cannot be a record field.

A scalar field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names followed by the name of a scalar field. A scalar field reference refers to a single data item (having a data type) and can be treated like a normal reference to a Fortran variable or array element.

You can use scalar field references in statement functions and in executable statements. However, they cannot be used in COMMON, SAVE, NAMELIST, or EQUIVALENCE statements, or as the control variable in an indexed DO-loop.

Type conversion rules for scalar field references are the same as those for variables and array elements.

An aggregate field reference consists of the name of a record (as specified in a RECORD statement) and zero or more levels of aggregate field names.

You can only assign an aggregate field to another aggregate field (record = record) if the records have the same structure. Intel® Fortran supports no other operations (such as arithmetic or comparison) on aggregate fields.

Intel Fortran requires qualification on all levels. While some languages allow omission of aggregate field names when there is no ambiguity as to which field is intended, Intel Fortran requires all aggregate field names to be included in references.

You can use aggregate field references in unformatted I/O statements; one I/O record is written no matter how many aggregate and array name references appear in the I/O list. You cannot use aggregate field references in formatted, namelist, and list-directed I/O statements.

You can use aggregate field references as actual arguments and record dummy arguments. The declaration of the dummy record in the subprogram must match the form of the aggregate field reference passed by the calling program unit; each structure must have the same number and types of fields in the same order. The order of map fields within a union declaration is irrelevant.

Records are passed by reference. Aggregate field references are treated like normal variables. You can use adjustable arrays in RECORD statements that are used as dummy arguments.

Examples

The following examples show record and field references. Consider the following structure declarations:

Structure DATE:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
STRUCTURE
```

Structure APPOINTMENT:

```
STRUCTURE /APPOINTMENT/
  RECORD /DATE/      APP_DATE
  STRUCTURE /TIME/   APP_TIME(2)
    INTEGER*1        HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20       APP_MEMO(4)
  LOGICAL*1          APP_FLAG
END STRUCTURE
```

The following RECORD statement creates a variable named NEXT_APP and a 10-element array named APP_LIST. Both the variable and each element of the array take the form of the structure APPOINTMENT.

```
RECORD /APPOINTMENT/    NEXT_APP,APP_LIST(10)
```

Each of the following examples of record and field references are derived from the previous structure declarations and RECORD statement:

Aggregate Field References

- The record NEXT_APP:

  ```
  NEXT_APP
  ```

- The field APP_DATE, a 4-byte array field in the record array APP_LIST(3):

  ```
  APP_LIST(3).APP_DATE
  ```

Scalar Field References

- The field APP_FLAG, a LOGICAL field of the record NEXT_APP:

  ```
  NEXT_APP.APP_FLAG
  ```

- The first character of APP_MEMO(1), a CHARACTER*20 field of the record NEXT_APP:

  ```
  NEXT_APP.APP_MEMO(1)(1:1)
  ```

NOTE. Because periods are used in record references to separate fields, you should avoid using relational operators (.EQ., .XOR.), logical constants (.TRUE., .FALSE.), and logical expressions (.AND., .NOT., .OR.) as field names in structure declarations. Dots can also be used instead of % to separate fields of a derived type.

Consider the following example:

```
module mod
   type T1_t
     integer :: i
   end type T1_t
  type T2_t
     type (T1_t) :: eq
     integer     :: i
  end type T2_t
  interface operator (.eq.)
     module procedure eq_func
  end interface operator (.eq.)
contains
  function eq_func(t2, i) result (rslt)
  type(T2_t), intent (in) :: t2
  integer,    intent (in) :: i
      rslt = t2%eq%i + i
  end function eq_func
end module mod
use mod
type(T2_t) :: t2
integer    :: i
  t2%eq%i = 0
  t2%i    = -10
  i       = -10
  print *, t2.eq.i, (t2).eq.i
end
```

In this case, the reference "t2.eq.i" prints 0. The reference "(t2).eq.i" will invoke eq_func and will print -10.

See Also

- Record Structures
- RECORD statement
- STRUCTURE
- UNION

Building Applications for details on alignment of data

Aggregate Assignment

For aggregate assignment statements, the variable and expression must have the same structure as the aggregate they reference.

The aggregate assignment statement assigns the value of each field of the aggregate on the right of an equal sign to the corresponding field of the aggregate on the left. Both aggregates must be declared with the same structure.

Examples

The following example shows valid aggregate assignments:

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE
RECORD /DATE/ TODAY, THIS_WEEK(7)
STRUCTURE /APPOINTMENT/
  ...
  RECORD /DATE/ APP_DATE
END STRUCTURE
RECORD /APPOINTMENT/ MEETING
DO I =  1,7
  CALL GET_DATE (TODAY)
  THIS_WEEK(I) =     TODAY
  THIS_WEEK(I).DAY = TODAY.DAY + 1
END DO
MEETING.APP_DATE =   TODAY
```

Additional Character Sets

59

This topic contains information about the following additional character sets:

- Character and Key Code Charts for Windows* OS
- The ASCII Character Set for Linux* OS and Mac OS* X

Character and Key Code Charts for Windows* OS

This topic contains the ASCII and ANSI character code charts, and the Key code charts that are available on Windows* OS.

ASCII Character Codes for Windows* Systems

The ASCII character code charts contain the decimal and hexadecimal values of the extended ASCII (American Standards Committee for Information Interchange) character set. The extended character set includes the ASCII character set (Chart 1) and 128 other characters for graphics and line drawing (Chart 2), often called the "IBM* character set".

ASCII Character Codes Chart 1 (W*32, W*64)

ASCII Character Codes Chart 2: IBM* Character Set (W*32, W*64)

ANSI Character Codes for Windows* Systems

The ANSI character code chart lists the extended character set of most of the programs used by Windows* systems. The codes of the ANSI (American National Standards Institute) character set from 32 through 126 are displayable characters from the ASCII character set. The ANSI characters displayed as solid blocks are undefined characters and may appear differently on output devices.

ANSI Character Codes Chart (W*32, W*64)

Key Codes for Windows* Systems

Some keys, such as function keys, cursor keys, and ALT+KEY combinations, have no ASCII code. When a key is pressed, a microprocessor within the keyboard generates an "extended scan code" of two bytes.

The first (low-order) byte contains the ASCII code, if any. The second (high-order) byte has the scan code--a unique code generated by the keyboard when a key is either pressed or released. Because the extended scan code is more extensive than the standard ASCII code, programs can use it to identify keys which do not have an ASCII code.

For more details on key codes, see:

- Key Codes Chart 1
- Key Codes Chart 2

Key Codes Chart 1 (W*32, W*64)

Key Codes Chart 2 (W*32, W*64)

ASCII Character Set for Linux* OS and Mac OS* X

This topic describes the ASCII character set that is available on Linux* OS and Mac OS* X.

The ASCII character set contains characters with decimal values 0 through 127. The first half of each of the numbered columns identifies the character as you would enter it on a terminal or as you would see it on a printer. Except for SP and HT, the characters with names are nonprintable. In the figure, the characters with names are defined as follows:

NUL	Null	DC1	Device Control 1 (XON)
SOH	Start of Heading	DC2	Device Control 2
STX	Start of Text	DC3	Device Control 1 (XOFF)
ETX	End of Text	DC4	Device Control 4
EOT	End of Transmission	NAK	Negative Acknowledge
ENQ	Enquiry	SYN	Synchronous Idle
ACK	Acknowledge	ETB	End of Transmission Block
BEL	Bell	CAN	Cancel
BS	Backspace	EM	End of Medium
HT	Horizontal Tab	SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tab	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	SP	Space

DLE	Data Link Escape	DEL	Delete

2220

The remaining half of each column identifies the character by the binary value of the byte; the value is stated in three radixes—octal, decimal, and hexadecimal. For example, the uppercase letter A has, under ASCII conventions, a storage value of hexadecimal 41 (a bit configuration of 01000001), equivalent to 101 in octal notation and 65 in decimal notation.

Figure 35: ASCII Character Set (L*X, M*X)

Data Representation Models

60

Several of the numeric intrinsic functions are defined by a model set for integers (for each intrinsic kind used) and reals (for each real kind used). The bit functions are defined by a model set for bits (binary digits).

The following intrinsic functions provide information on the data representation models:

Intrinsic function	Model	Value returned
BIT_SIZE	Bit	The number of bits (s) in the bit model
DIGITS	Integer or Real	The number of significant digits in the model for the argument
EPSILON	Real	The number that is almost negligible when compared to one
EXPONENT	Real	The value of the exponent part of a real argument
FRACTION	Real	The fractional part of a real argument
HUGE	Integer or Real	The largest number in the model for the argument
MAXEXPONENT	Real	The maximum exponent in the model for the argument
MINEXPONENT	Real	The minimum exponent in the model for the argument
NEAREST	Real	The nearest different machine-representable number in a given direction
PRECISION	Real	The decimal precision (real or complex) of the argument
RADIX	Integer or Real	The base of the model for the argument

Intrinsic function	Model	Value returned
RANGE	Integer or Real	The decimal exponent range of the model for the argument
RRSPACING	Real	The reciprocal of the relative spacing near the argument
SCALE	Real	The value of the exponent part (of the model for the argument) changed by a specified value
SET_EXPONENT	Real	The value of the exponent part (of the model for the argument) set to a specified value
SPACING	Real	The value of the absolute spacing of model numbers near the argument
TINY	Real	The smallest positive number in the model for the argument

For more information on the range of values for each data type (and kind), see *Building Applications*.

This appendix discusses the following topics:

- The model for Integer Data
- The model for Real Data
- The model for Bit Data

Model for Integer Data

In general, the model set for integers is defined as follows:

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

The following values apply to this model set:

- i is the integer value.
- s is the sign (either +1 or -1).
- q is the number of digits (a positive integer).
- r is the radix (an integer greater than 1).
- w_k is a nonnegative number less than r.

The model for INTEGER(4) follows:

$$i = s \times \sum_{k=1}^{31} w_k \times 2^{k-1}$$

The following example shows the general integer model for i = -20 using a base (r) of 2:

$$i = (-1) \times (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4)$$
$$i = (-1) \times (4 + 16)$$
$$i = -1 \times 20$$
$$i = -20$$

Model for Real Data

The model set for reals, in general, is defined as one of the following:

$$x = 0$$

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

The following values apply to this model set:

- x is the real value.
- s is the sign (either +1 or -1).
- b is the base (real radix; an integer greater than 1; b = 2 in Intel® Fortran).

- p is the number of mantissa digits (an integer greater than 1). The number of digits differs depending on the real format, as follows:

REAL(4)	IEEE S_floating	24
REAL(8)	IEEE T_floating	53
REAL(16)	IEEE X_floating	113

- e is an integer in the range e_{min} to e_{max} inclusive. This range differs depending on the real format, as follows:

		e_{min}	e_{max}
REAL(4)	IEEE S_floating	-125	128
REAL(8)	IEEE T_floating	-1021	1024
REAL(16)	IEEE X_floating	-16381	16384

- f_k is a nonnegative number less than b (f_1 is also nonzero).

For x = 0, its exponent e and digits f_k are defined to be zero.

The model set for single-precision real (REAL(4)) is defined as one of the following:

$$x = 0$$

$$x = s \times 2^{e} \times \left[1/2 + \sum_{k=2}^{24} f_k \times 2^{-k}\right], -125 \le e \le 128$$

The following example shows the general real model for x = 20.0 using a base (b) of 2:

$$x = 1 \times 2^{5} \times (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3})$$
$$x = 1 \times 32 \times (.5 + .125)$$
$$x = 32 \times (.625)$$
$$x = 20.0$$

Model for Bit Data

The model set for bits (binary digits) interprets a nonnegative scalar data object of type integer as a sequence, as follows:

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

The following values apply to this model set:

- j is the integer value.
- s is the number of bits.
- w_k is a bit value of 0 or 1.

The bits are numbered from right to left beginning with 0.

The following example shows the bit model for j = 1001 (integer 9) using a bit number (s) of 4:

$$\begin{array}{cccc} 1 & 0 & 0 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ w_3 & w_2 & w_1 & w_0 \end{array}$$

$$j = (w_0 \times 2^0) + (w_1 \times 2^1) + (w_2 \times 2^2) + (w_3 \times 2^3)$$

$$j = 1 + 0 + 0 + 8$$

$$j = 9$$

Run-Time Library Routines

61

Intel® Fortran provides the following run-time library routines, which are summarized in this appendix:

- Module routines
- OpenMP* Fortran routines

Module Routines

Intel® Fortran provides library modules containing the following routines:

- Routines that help you write programs for graphics, QuickWin, and other applications (in modules IFQWIN, IFLOGM, and IFCORE):
  - QuickWin routines (W*32, W*64)
  - Graphics routines (W*32, W*64)
  - Dialog routines (W*32, W*64)
  - Miscellaneous run-time routines
- Routines systems that help you write programs using Component Object Model (COM) and Automation servers (in modules IFCOM and IFAUTO):
  - COM routines (W*32, W*64)
  - AUTO routines (W*32, W*64)
- Portability routines that help you port your programs to or from other systems, or help you perform basic I/O to serial ports on Windows* systems (in module IFPORT).
- National Language Support routines that help you write foreign language programs for international markets (in module IFNLS). These routines are only available on Windows* systems.
- POSIX routines that help you write Fortran programs that comply with the POSIX* Standard (in module IFPOSIX).

When you include the statement USE *module-name* in your program, these library routines are automatically linked to your program if called.

You can restrict what is accessed from a USE module by adding ONLY clauses to the USE statement.

See Also

- Run-Time Library Routines

- USE

Building Applications: Calling Library Routines

OpenMP* Fortran Routines

The following table summarizes the OpenMP Fortran API run-time library routines you can use for directed parallel decomposition. These routines are all external procedures.

To use these routines, you must add a USE OMP_LIB statement to the program unit containing the routine.

Table 687: Summary of OpenMP Fortran Parallel Routines

Name	Description
OMP_SET_NUM_THREADS	Sets the number of threads to use for the next parallel region.
OMP_GET_NUM_THREADS	Gets the number of threads currently in the team executing the parallel region from which the routine is called.
OMP_GET_MAX_THREADS	Gets the maximum value that can be returned by calls to the OMP_GET_NUM_THREADS function.
OMP_GET_THREAD_NUM	Gets the thread number, within the team, in the range from zero to OMP_GET_NUM_THREADS minus one.
OMP_GET_NUM_PROCS	Gets the number of processors that are available to the program.
OMP_IN_PARALLEL	Informs whether or not a region is executing in parallel.
OMP_SET_DYNAMIC	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
OMP_GET_DYNAMIC	Informs whether or not dynamic thread adjustment is enabled.
OMP_SET_NESTED	Enables or disables nested parallelism.

Name	Description
OMP_GET_NESTED	Informs whether or not nested parallelism is enabled.
OMP_INIT_LOCK	Initializes a lock to be used in subsequent calls.
OMP_DESTROY_LOCK	Disassociates a lock variable from any locks.
OMP_SET_LOCK	Makes the executing thread wait until the specified lock is available.
OMP_UNSET_LOCK	Releases the executing thread from ownership of a lock.
OMP_TEST_LOCK	Tries to set the lock associated with a lock variable.
OMP_INIT_NEST_LOCK	Initializes a nested lock for use in subsequent calls.
OMP_DESTROY_NEST_LOCK	Disassociates a lock variable from a nested lock.
OMP_SET_NEST_LOCK	Makes the executing thread wait until the specified nested lock is available.
OMP_UNSET_NEST_LOCK	Releases the executing thread from ownership of a nested lock if the nesting count is zero.
OMP_TEST_NEST_LOCK	Tries to set the nested lock associated with a lock variable.
OMP_GET_WTIME	Returns a double-precision value equal to the elapsed wallclock time (in seconds) relative to an arbitrary reference time.
OMP_GETWTICK	Returns a double-precision value equal to the number of seconds between successive clock ticks.

Intel® Fortran Extensions:

Name	Description
KMP_GET_STACKSIZE_S[1]	Returns the number of bytes that will be allocated for each parallel thread to use as its private stack.
KMP_SET_STACKSIZE_S[2]	Sets the number of bytes that will be allocated for each parallel thread to use as its private stack.
KMP_GET_BLOCKTIME	Returns the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping.
KMP_SET_BLOCKTIME	Sets the number of milliseconds that a thread should wait, after completing the execution of a parallel region, before sleeping.
KMP_MALLOC	Allocates a memory block of a specified size (in bytes) from the thread-local heap.
KMP_CALLOC	Allocates an array of a specified number of elements and size from the thread-local heap.
KMP_REALLOC	Reallocates a memory block at a specified address and of a specified size from the thread-local heap.
KMP_FREE	Frees a memory block at a specified address from the thread-local heap.

[1] For backwards compatibility, this can also be specified as KMP_GET_STACKSIZE.

[2] For backwards compatibility, this can also be specified as KMP_SET_STACKSIZE.

For more information on a specific routine, see the appropriate reference page; for example, for more information on OMP_SET_LOCK, see `omp_set_lock(3f)`.

See Also

- Run-Time Library Routines
- OpenMP Fortran Compiler Directives

Optimizing Applications: Intel Extension Routines/Functions

Summary of Language Extensions

62

This appendix summarizes the Intel® Fortran language extensions to the ANSI/ISO Fortran 95 Standard.

Most extensions are available on all supported operating systems. However, some extensions are limited to one or more platforms. If an extension is limited, it is labeled.

Extensions related to the following topics are discussed:

- Source Forms
- Names
- Character Sets
- Intrinsic Data Types
- Constants
- Expressions and Assignment
- Specification Statements
- Execution Control
- Program Units and Procedures
- Compilation Control Statements
- Built-In Functions
- I/O Statements
- I/O Formatting
- File Operation Statements
- Compiler Directives
- Intrinsic Procedures
- Additional Language Features
- Run-Time Library Routines

Language Extensions: Source Forms

The following are extensions to the methods and rules for source forms:

- Tab-formatting as a method to code lines
- The letter D as a debugging statement indicator in column 1 of fixed or tab source form

- An optional statement field width of 132 columns for fixed or tab source form
- An optional sequence number field for fixed source form
- Up to 511 continuation lines in a source program

Language Extensions: Names

The following are extensions to the Fortran 90 rules for names (see names):

- Names can contain up to 63 characters
- The dollar sign ($) is a valid character in names, and can be the first character

Language Extensions: Character Sets

The following are extensions to the Fortran 90 character set:

- The Tab (<Tab>) character (see Character Sets)
- ASCII Character Code Chart 2 -- IBM* Character Set
- ANSI Character Code Chart
- Key Code Charts

Language Extensions: Intrinsic Data Types

The following are data-type extensions:

BYTE	INTEGER*1	REAL*16
DOUBLE COMPLEX	INTEGER*2	COMPLEX*8
LOGICAL*1	INTEGER*4	COMPLEX*16
LOGICAL*2	INTEGER*8	COMPLEX*32
LOGICAL*4	REAL*4	
LOGICAL*8	REAL*8	

See Also

- Summary of Language Extensions
- Intrinsic Data Types

Language Extensions: Constants

Hollerith constants are allowed as an extension.

C Strings are allowed as extensions in character constants.

Language Extensions: Expressions and Assignment

When operands of different intrinsic data types are combined in expressions, conversions are performed as necessary (see Data Type of Numeric Expressions).

Binary, octal, hexadecimal, and Hollerith constants can appear wherever numeric constants are allowed.

The following are extensions allowed in logical expressions:

- .XOR. as a synonym for .NEQV.
- Integers as valid logical items
- Logical operators applied to integers bit-by-bit

Language Extensions: Specification Statements

The following specification attributes and statements are extensions:

- AUTOMATIC attribute and statement
- PROTECTED attribute and statement
- STATIC attribute and statement
- VOLATILE attribute and statement

A double colon is now optional for the INTRINSIC, SAVE, STATIC, AUTOMATIC, EXTERNAL, and VOLATILE statements.

Language Extensions: Execution Control

The following control statements are extensions to Fortran 95:

- ASSIGN
- Assigned GO TO
- PAUSE

These are older Fortran features that have been deleted in Fortran 95. Intel® Fortran fully supports these features.

Language Extensions: Program Units and Procedures

The following program units and statement are extensions:

- Intrinsic modules
- The IMPORT statement

Language Extensions: Compilation Control Lines and Statements

The following line option and statement are extensions that can influence compilation:

- [/[NO]LIST], which can be specified in an INCLUDE line
- The OPTIONS statement

Language Extensions: Built-In Functions

The following built-in functions are extensions:

- %VAL, %REF, and %LOC, which facilitate references to non-Fortran procedures
- %FILL, which can be used in record structure type definitions

Language Extensions: I/O Statements

The following I/O statements are extensions:

- The ACCEPT statement
- The FLUSH statement
- The REWRITE statement
- The TYPE statement, which is a synonym for the PRINT statement
- The WAIT statement

Language Extensions: I/O Formatting

The following are extensions allowed in I/O Formatting:

- The Q edit descriptor
- The dollar sign ($) edit descriptor and carriage-control character
- The backslash (\) edit descriptor
- The ASCII NUL carriage-control character
- Variable format expressions
- The H edit descriptor

  This is an older Fortran feature that has been deleted in Fortran 95. Intel® Fortran fully supports this feature.

Language Extensions: File Operation Statements

The following statement specifiers and statements are extensions:

- CLOSE statement specifiers:
  - STATUS values: 'SAVE' (as a synonym for 'KEEP'), 'PRINT', 'PRINT/DELETE', 'SUBMIT', 'SUBMIT/DELETE'
  - DISPOSE (or DISP)
- DELETE statement
- INQUIRE statement specifiers:
  - ASYNCHRONOUS
  - BINARY (W*32, W*64)
  - BLOCKSIZE
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - FORM values: 'UNKNOWN', 'BINARY' (W*32, W*64)
  - IOFOCUS (W*32, W*64)

- MODE as a synonym for ACTION
- ORGANIZATION
- PENDING
- POS
- RECORDTYPE
- SHARE (W*32, W*64)

See also INQUIRE Statement.

- OPEN statement specifiers:
  - ACCESS values: 'APPEND'
  - ASSOCIATEVARIABLE
  - ASYNCHRONOUS
  - BLOCKSIZE
  - BUFFERCOUNT
  - BUFFERED
  - CARRIAGECONTROL
  - CONVERT
  - DEFAULTFILE
  - DISPOSE (or DISP)
  - FORM value: 'BINARY' (W*32, W*64)
  - IOFOCUS (W*32, W*64)
  - MAXREC
  - MODE as a synonym for ACTION
  - NAME as a synonym for FILE
  - NOSHARED
  - ORGANIZATION
  - READONLY
  - RECORDSIZE as a synonym for RECL
  - RECORDTYPE
  - SHARE (W*32, W*64)
  - SHARED
  - TITLE (W*32, W*64)

- TYPE as a synonym for STATUS
- USEROPEN

See also OPEN Statement.

Language Extensions: Compiler Directives

The following General Directives are extensions:

- ALIAS
- ASSUME_ALIGNED
- ATTRIBUTES
- DECLARE and NODECLARE
- DEFINE and UNDEFINE
- DISTRIBUTE POINT
- FIXEDFORMLINESIZE
- FREEFORM and NOFREEFORM
- IDENT
- IF and IF DEFINED
- INTEGER
- IVDEP
- LOOP COUNT
- MEMORYTOUCH (i64 only)
- MEMREF_CONTROL (i64 only)
- MESSAGE
- OBJCOMMENT
- OPTIMIZE and NOOPTIMIZE
- OPTIONS
- PACK
- PARALLEL and NOPARALLEL (loop)
- PREFETCH and NOPREFETCH
- PSECT
- REAL

- STRICT and NOSTRICT
- SWP and NOSWP (i64 only)
- UNROLL and NOUNROLL
- UNROLL_AND_JAM and NOUNROLL_AND_JAM
- VECTOR ALIGNED and VECTOR UNALIGNED
- VECTOR ALWAYS and NOVECTOR
- VECTOR TEMPORAL and VECTOR NONTEMPORAL (i32, i64em)

The following OpenMP* Fortran directives are extensions:

- ATOMIC
- BARRIER
- CRITICAL
- DO
- FLUSH
- MASTER
- ORDERED
- PARALLEL
- PARALLEL DO
- PARALLEL SECTIONS
- PARALLEL WORKSHARE
- SECTIONS
- SINGLE
- TASK
- TASKWAIT
- THREADPRIVATE
- WORKSHARE

Language Extensions: Intrinsic Procedures

The following intrinsic procedures are extensions available on all platforms:

Table 689: A to D

ACOSD	BIAND	COMMAND_ARGUMENT_COUNT	DCMPLX

ACOSH	BIEOR	COSD	DCONJG
AIMIN0	BITEST	COTAND	DCOSD
AJMAX0	BIOR	CQABS	DCOTAN
AJMIN0	BJTEST	CQCOS	DCOTAND
AKMAX0	BKTEST	CQEXP	DERF
AKMIN0	BMOD	CQLOG	DERFC
AND	BMVBITS	CQSIN	DFLOAT
ASIND	BNOT	CQSQRT	DFLOTI
ASINH	BSHFT	CQTAN	DFLOTJ
ATAN2D	BSHFTC	CTAN	DFLOTK
ATAND	BSIGN	DACOSD	DIMAG
ATANH	CACHESIZE	DACOSH	DNUM
BABS	CDABS	DASIND	DREAL
BADDRESS	CDCOS	DASINH	DSHIFTL
BBCLR	CDEXP	DATAN2D	DSHIFTR
BBITS	CDLOG	DATAND	DSIND
BBSET	CDSIN	DATAN	DTAND
AIMAX0	BIOR	COTAN	
BBTEST	CDSQRT	DATE	
BDIM	CDTAN	DBLEQ	

Table 690: E to I

EOF	HIEOR	IIDINT	ININT
ERF	HIOR	IIDNNT	INOT

ERFC	HIXOR	IIEOR	INT1
ERRSNS	HMOD	IIFIX	INT2
EXIT	HMVBITS	IINT	INT4
FLOATI	HNOT	IIOR	INT8
FLOATJ	HSHFT	IIQINT	INT_PTR_KIND
FLOATK	HSHFTC	IIQNNT	INUM
FP_CLASS	HSIGN	IISHFT	IQINT
FREE	HTEST	IISHFTC	IQNINT
GETARG	IADDR	IISIGN	IS_IOSTAT_END
GET_COMMAND	IARG	IIXOR	IS_IOSTAT_EOR
GET_COMMAND ARGUMENT	IARGC	IJINT	ISHA
GET_ENVIRONMENT_VARIABLE	IBCHNG	ILEN	ISHC
HABS	IDATE	IMAG	ISHL
HBCLR	IIABS	IMAX0	ISNAN
HBITS	IIAND	IMAX1	IXOR
HBSET	IIBCLR	IMIN0	IZEXT
HDIM	IIBITS	IMIN1	
HFIX	IIBSET	IMOD	
HIAND	IIDIM	IMVBITS	

Table 691: J to P

JFIX	JMAX0	KIEOR	LEADZ
JIABS	JMAX1	KIFIX	LOC

JIAND	JMIN0	KINT	LSHIFT
JIBCLR	JMIN1	KIOR	LSHFT
JIBITS	JMOD	KIQINT	MALLOC
JIBSET	JMVBITS	KIQNNT	MCLOCK
JIDIM	JNINT	KISHFT	MM_PREFETCH
JIDINT	JNOT	KISHFTC	MOVE_ALLOC
JIDNNT	JNUM	KISIGN	MULT_HIGH
JIEOR	JZEXT	KMAX0	MULT_HIGH_SIGNED
JIFIX	KDIM	KMAX1	NARGS
JINT	KIABS	KMIN0	NEW_LINE
JIOR	KIAND	KMIN1	NUMARG
JIQINT	KIBCLR	KMOD	OR
JIQNNT	KIBITS	KMVBITS	POPCNT
JISHFT	KIBSET	KNINT	POPPAR
JISHFTC	KIDIM	KNOT	
JISIGN	KIDINT	KNUM	
JIXOR	KIDNNT	KZEXT	

Table 692: Q to Z

QABS	QCOSH	QNINT	SHIFTL
QACOS	QCOTAN	QNUM	SHIFTR
QACOSD	QCOTAND	QREAL	SIND
QACOSH	QDIM	QSIGN	SIZEOF
QARCOS	QERF	QSIN	SNGLQ

QASIN	QERFC	QSIND	TAND
QASIND	QEXP	QSINH	TIME
QASINH	QEXT	QSQRT	TRAILZ
QATAN	QEXTD	QTAN	XOR
QATAN2	QFLOAT	QTAND	ZABS
QATAN2D	QIMAG	QTANH	ZCOS
QATAND	QINT	RAN	ZEXP
QATANH	QLOG	RANDU	ZEXT
QCMPLX	QLOG10	RNUM	ZLOG
QCONJG	QMAX1	RSHIFT	ZSIN
QCOS	QMIN1	RSHFT	ZSQRT
QCOSD	QMOD	SECNDS	ZTAN

The argument KIND is an extension available in the following intrinsic procedures:

ACHAR	INDEX	MAXLOC	SIZE
COUNT	LBOUND	MINLOC	UBOUND
IACHAR	LEN	SCAN	VERIFY
ICHAR	LEN_TRIM	SHAPE	

Language Extensions: Additional Language Features

The following are language extensions that facilitate compatibility with other versions of Fortran:

- DEFINE FILE statement
- ENCODE and DECODE statements
- FIND statement
- The INTERFACE TO statement

- FORTRAN 66 Interpretation of the EXTERNAL statement
- An alternative syntax for the PARAMETER statement
- VIRTUAL statement
- AND, OR, XOR, IMAG, LSHIFT, RSHIFT intrinsics (see the *A to Z Reference*)
- An alternative syntax for octal and hexadecimal constants
- An alternative syntax for an I/O record specifier
- An alternate syntax for the DELETE statement
- An alternative form for namelist external records
- The integer POINTER statement
- Record structures

Language Extensions: Run-Time Library Routines

The following run-time library routines are available as extensions:

- Module routines
- OpenMP* Fortran routines

A to Z Reference

63

This section contains the following:

- Language Summary Tables

  This section organizes the functions, subroutines, and statements available in Intel® Fortran by the operations they perform. You can use the tables to locate a particular routine for a particular task.
- The descriptions of all Intel Fortran statements, intrinsics, directives, and module library routines, which are listed in alphabetical order.

  In the description of routines, pointers and handles are INTEGER(4) on IA-32 architecture and INTEGER(8) on Intel® 64 architecture and IA-64 architecture.

The Fortran compiler understands statements and intrinsic functions in your program without any additional information, such as that provided in modules.

However, modules must be included in programs that contain the following routines:

- Quickwin routines and graphics routines (W*32, W*64)

  These routines require a USE IFQWIN statement to include the library and graphics modules.
- Portability routines and serial port I/O routines

  These routines require a USE IFPORT statement to access the portability library. The serial port I/O routines are only available on Windows* systems.
- NLS routines (W*32, W*64)

  These routines require a USE IFNLS statement to access the NLS library.
- POSIX* routines

  These routines require a USE IFPOSIX statement to access the POSIX library.
- Dialog routines (W*32, W*64)

  These routines require a USE IFLOGM statement to access the dialog library.
- Component Object Module (COM) routines (W*32, W*64)

  These routines require a USE IFCOM statement to access the COM library.
- Automation server routines (W*32, W*64)

  These routines require a USE IFAUTO statement to access the AUTO library.
- Miscellaneous Run-Time Routines

  Most of these routines require a USE IFCORE statement to obtain the proper interfaces.

Whenever required, these USE module statements are prominent in the *A to Z Reference*.

In addition to the appropriate USE statement, for some routines you must specify the types of libraries to be used when linking.

Language Summary Tables

The Fortran procedures and statements have been organized into the following tables:

- Statements for Program Unit Calls and Definition
- Statements Affecting Variables
- Statements for Input and Output
- Compiler Directives
- Program Control Statements and Procedures
- Inquiry Intrinsic Procedures
- Random Number Intrinsic Procedures
- Date and Time Intrinsic Subroutines
- Keyboard and Speaker Library Routines
- Statements and Intrinsic Procedures Memory Allocation and Deallocation
- Intrinsic Functions for Arrays
- Intrinsic Functions for Numeric and Type Conversion
- Trigonometric, Exponential, Root, and Logarithmic Intrinsic Procedures
- Intrinsic Functions for Floating-Point Inquiry and Control
- Character Intrinsic Functions
- Intrinsic Procedures for Bit Operation and Representation
- QuickWin Library Routines
- Graphics Library Routines
- Portability Library Routines
- National Language Standard Library Routines
- POSIX* Library Routines
- Dialog Library Routines
- COM and Automation Library Routines
- Miscellaneous Run-Time Library Routines
- Functions Not Allowed as Actual Arguments

See Also

- A to Z Reference
- Statements for Program Unit Calls and Definitions
- Statements Affecting Variables
- Statements for Input and Output
- Compiler Directives
- Program Control Statements and Procedures
- Inquiry Intrinsic Functions
- Random Number Intrinsic Procedures
- Date and Time Intrinsic Subroutines
- Keyboard and Speaker Library Routines
- Statements and Intrinsic Procedures for Memory Allocation and Deallocation
- Intrinsic Functions for Arrays
- Intrinsic Functions for Numeric and Type Conversion
- Trigonometric, Exponential, Root, and Logarithmic Intrinsic Procedures
- Intrinsic Functions for Floating-Point Inquiry and Control
- Character Intrinsic Functions
- Intrinsic Procedures for Bit Operation and Representation
- QuickWin Library Routines (W*32, W*64)
- Graphics Library Routines (W*32, W*64)
- Portability Library Routines
- National Language Support Library Routines (W*32, W*64)
- POSIX* Library Procedures
- Dialog Library Routines (W*32, W*64)
- COM and Automation Library Routines (W*32, W*64)
- Miscellaneous Run-Time Library Routines
- Intrinsic Functions Not Allowed as Actual Arguments
- Argument Keywords in Intrinsic Procedures
- Program Units and Procedures

Statements for Program Unit Calls and Definitions

The following table lists statements used for program unit calls and definition.

Name	Description
BLOCK DATA	Identifies a block-data subprogram.
CALL	Executes a subroutine.
COMMON	Delineates variables shared between program units.
CONTAINS	Identifies start of a module within a host module.
ENTRY	Specifies a secondary entry point to a subroutine or external function.
EXTERNAL	Declares a user-defined subroutine or function to be passable as an argument.
FUNCTION	Identifies a program unit as a function.
INCLUDE	Inserts the contents of a specified file into the source file.
INTERFACE	Specifies an explicit interface for external functions and subroutines.
INTRINSIC	Declares a predefined function.
MODULE	Identifies a module program unit.
PROGRAM	Identifies a program unit as a main program.
RETURN	Returns control to the program unit that called a subroutine or function.
SUBROUTINE	Identifies a program unit as a subroutine.
USE	Gives a program unit access to a module.

Statements Affecting Variables

The following table lists statements that affect variable.

Name	Description
AUTOMATIC	Declares a variable on the stack, rather than at a static memory location.
BYTE	Specifies variables as the BYTE data type; BYTE is equivalent to INTEGER(1).
CHARACTER	Specifies variables as the CHARACTER data type.
COMPLEX	Specifies variables as the COMPLEX data type.
DATA	Assigns initial values to variables.
DIMENSION	Identifies a variable as an array and specifies the number of elements.
DOUBLE COMPLEX	Specifies variables as the DOUBLE COMPLEX data type, equivalent to COMPLEX(8).
DOUBLE PRECISION	Specifies variables as the DOUBLE-PRECISION real data type, equivalent to REAL(8).
EQUIVALENCE	Specifies that two or more variables or arrays share the same memory location.
IMPLICIT	Specifies the default typing for real and integer variables and functions.
INTEGER	Specifies variables as the INTEGER data type.
LOGICAL	Specifies variables as the LOGICAL data type.
MAP	Within a UNION statement, delimits a group of variable type declarations that are to be ordered contiguously within memory.
NAMELIST	Declares a group name for a set of variables to be read or written in a single statement.
PARAMETER	Equates a constant expression with a name.

Name	Description
PROTECTED	Specifies limitations on the use of module entities.
REAL	Specifies variables as the REAL data type.
RECORD	Declares one or more variables of a user-defined structure type.
SAVE	Causes variables to retain their values between invocations of the procedure in which they are defined.
STATIC	Declares a variable is in a static memory location, rather than on the stack.
STRUCTURE	Defines a new variable type, composed of a collection of other variable types.
TYPE	Defines a new variable type, composed of a collection of other variable types.
UNION	Within a structure, causes two or more maps to occupy the same memory locations.
VOLATILE	Specifies that the value of an object is totally unpredictable based on information available to the current program unit.

Statements for Input and Output

The following table lists statements used for input and output.

Name	Procedure Type	Description
ACCEPT	Statement	Similar to a formatted, sequential READ statement.
BACKSPACE	Statement	Positions a file to the beginning of the previous record.

Name	Procedure Type	Description
CLOSE	Statement	Disconnects the specified unit.
DELETE	Statement	Deletes a record from a relative file.
ENDFILE	Statement	Writes an end-of-file record or truncates a file.
EOF	Intrinsic Function	Checks for end-of-file record. .TRUE. if at or past end-of-file.
INQUIRE	Statement	Returns the properties of a file or unit.
OPEN	Statement	Associates a unit number with an external device or file.
PRINT (or TYPE)	Statement	Displays data on the screen.
READ	Statement	Transfers data from a file to the items in an I/O list.
REWIND	Statement	Repositions a file to its first record.
REWRITE	Statement	Rewrites the current record.
WRITE	Statement	Transfers data from the items in an I/O list to a file

Compiler Directives

The following table lists available compiler directives.

Each general directive name is preceded by the prefix *c*DEC$; for example, *c*DEC$ ALIAS. Each OpenMP* Fortran directive name is preceded by the prefix *c*$OMP; for example, *c*$OMP ATOMIC. The *c* in either can be a c, C, *, or ! in fixed-form source code; only ! in free-form source code.

Table 697: General Directives

Name	Description
ALIAS	Specifies an alternate external name to be used when referring to external subprograms.
ASSUME_ALIGNED	Specifies that an entity in memory is aligned.
ATTRIBUTES	Applies attributes to variables and procedures.
DECLARE	Generates warning messages for undeclared variables.
DEFINE	Creates a variable whose existence can be tested during conditional compilation.
DISTRIBUTE POINT	Specifies distribution for a DO loop.
ELSE	Marks the beginning of an alternative conditional-compilation block to an IF directive construct.
ELSEIF	Marks the beginning of an alternative conditional-compilation block to an IF directive construct.
ENDIF	Marks the end of a conditional-compilation block.
FIXEDFORMLINESIZE	Sets fixed-form line length. This directive has no effect on freeform code.
FREEFORM	Uses freeform format for source code.
IDENT	Specifies an identifier for an object module.
IF	Marks the beginning of a conditional-compilation block.
IF DEFINED	Marks the beginning of a conditional-compilation block.
INTEGER	Selects default integer size.

Name	Description
IVDEP	Assists the compiler's dependence analysis of iterative DO loops.
LOOP COUNT	Specifies the loop count for a DO loop; this assists the optimizer.
MEMORYTOUCH[1]	Ensures that a specific memory location is updated dynamically.
MEMREF_CONTROL[1]	Lets you provide cache hints on prefetches, loads, and stores.
MESSAGE	Sends a character string to the standard output device.
NODECLARE	(Default) Turns off warning messages for undeclared variables.
NOFREEFORM	(Default) Uses standard FORTRAN 77 code formatting column rules.
NOPARALLEL	Disables auto-parallelization for an immediately following DO loop.
NOOPTIMIZE	Disables optimizations.
NOPREFETCH	Disables a data prefetch from memory.
NOSTRICT	(Default) Disables a previous STRICT directive.
NOSWP[1]	Disables software pipelining for a DO loop.
NOUNROLL	Disables the unrolling of a DO loop.
NOUNROLL_AND_JAM	Disables loop unrolling and jamming.
NOVECTOR[2]	Disables vectorization of a DO loop.
OBJCOMMENT	Specifies a library search path in an object file.

Name	Description
OPTIMIZE	Enables optimizations.
OPTIONS	Controls whether fields in records and data items in common blocks are naturally aligned or packed on arbitrary byte boundaries.
PACK	Specifies the memory starting addresses of derived-type items.
PARALLEL	Enables auto-parallelization for an immediately following DO loop.
PREFETCH	Enables a data prefetch from memory.
PSECT	Modifies certain characteristics of a common block.
REAL	Selects default real size.
STRICT	Disables Intel® Fortran features not in the language standard specified on the command line (Fortran 95 or Fortran 90).
SWP[1]	Enables software pipelining for a DO loop.
UNDEFINE	Removes a symbolic variable name created with the DEFINE directive.
UNROLL	Tells the compiler's optimizer how many times to unroll a DO loop.
UNROLL_AND_JAM	Enables loop unrolling and jamming.
VECTOR ALIGNED	Specifies that all data is aligned in a DO loop.
VECTOR ALWAYS	Enables vectorization of a DO loop.
VECTOR NONTEMPORAL[2]	Directs the compiler to use non-temporal (that is, streaming) stores.
VECTOR NOVECTOR	Disables vectorization of a DO loop.

Name	Description
VECTOR TEMPORAL[2]	Directs the compiler to use temporal (that is, non-streaming) stores.
VECTOR UNALIGNED	Specifies that no data is aligned in a DO loop.

[1]i64 only

[2]i32, i64em

To use the following directives, you must specify compiler option `-openmp` (Linux and Mac OS X) or `/Qopenmp` (Windows).

Table 698: OpenMP Fortran Directives

Name	Description
ATOMIC	Specifies that a specific memory location is to be updated dynamically.
BARRIER	Synchronizes all the threads in a team.
CRITICAL	Restricts access for a block of code to only one thread at a time.
DO	Specifies that the iterations of the immediately following DO loop must be executed in parallel.
FLUSH	Specifies synchronization points where the implementation must have a consistent view of memory.
MASTER	Specifies a block of code to be executed by the master thread of the team.
ORDERED	Specifies a block of code to be executed sequentially.
PARALLEL	Defines a parallel region.
PARALLEL DO	Defines a parallel region that contains a single DO directive.

Name	Description
PARALLEL SECTIONS	Defines a parallel region that contains SECTIONS directives.
PARALLEL WORKSHARE	Defines a parallel region that contains a single WORKSHARE directive.
SECTIONS	Specifies a block of code to be divided among threads in a team (a worksharing area).
SINGLE	Specifies a block of code to be executed by only one thread in a team.
TASK	Defines a task region.
TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
THREADPRIVATE	Makes named common blocks private to a thread but global within the thread.
WORKSHARE	Divides the work of executing a block of statements or constructs into separate units.

Program Control Statements and Procedures

The following table lists statements and a procedure that affect program control.

Table 699: Statements

Name	Description
CASE	Within a SELECT CASE structure, marks a block of statements that are executed if an associated value matches the SELECT CASE expression.
CONTINUE	Often used as the target of GOTO or as the terminal statement in a DO loop; performs no operation.

Name	Description
CYCLE	Advances control to the end statement of a DO loop; the intervening loop statements are not executed.
DO	Evaluates statements in the DO loop, through and including the ending statement, a specific number of times.
DO WHILE	Evaluates statements in the DO WHILE loop, through and including the ending statement, until a logical condition becomes .FALSE..
ELSE	Introduces an ELSE block.
ELSE IF	Introduces an ELSE IF block.
ELSEWHERE	Introduces an ELSEWHERE block.
END	Marks the end of a program unit.
END DO	Marks the end of a series of statements following a DO or DO WHILE statement.
END FORALL	Marks the end of a series of statements following a block FORALL statement.
END IF	Marks the end of a series of statements following a block IF statement.
END SELECT	Marks the end of a SELECT CASE statement.
END WHERE	Marks the end of a series of statements following a block WHERE statement.
EXIT	Leaves a DO loop; execution continues with the first statement that follows.
FORALL	Controls conditional execution of other statements.
GOTO	Transfers control to a specified part of the program.

Name	Description
IF	Controls conditional execution of other statements.
PAUSE	Suspends program execution and, optionally, executes operating-system commands.
SELECT CASE	Transfers program control to a block of statements, determined by a controlling argument.
STOP	Terminates program execution.
WHERE	Controls conditional execution of other statements.

Table 700: Procedure

Name	Description
EXIT	(Intrinsic Subroutine) Terminates the program, flushes and closes all open files, and returns control to the operating system.

The portability routines RAISEQQ, SIGNALQQ, and SLEEPQQ also supply this functionality.

Inquiry Intrinsic Functions

The following table lists inquiry intrinsic functions.

Name	Description
ALLOCATED	Determines whether an allocatable array is allocated.
ASSOCIATED	Determines whether a the first pointer argument and the second (optional) pointer argument are associated.
BIT_SIZE	Returns the number of bits in an integer type.
CACHESIZE[1]	Returns the size of a level of the memory cache.

Name	Description
COMMAND_ARGUMENT_COUNT	Returns the number of command arguments.
DIGITS	Returns number of significant digits for data of the same type as the argument.
EOF	Determines whether a file is at or beyond the end-of-file record.
EPSILON	Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as the argument.
HUGE	Returns the largest number that can be represented by numbers of type the argument.
IARGC	Returns the index of the last command-line argument.
INT_PTR_KIND	Returns the INTEGER KIND that will hold an address.
KIND	Returns the value of the kind parameter of the argument.
LBOUND	Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.
LEN	Returns the length of a character expression.
LOC	Returns the address of the argument.
MAXEXPONENT	Returns the largest positive decimal exponent for data of the same type as the argument.
MINEXPONENT	Returns the largest negative decimal exponent for data of the same type as the argument.

Name	Description
NARGS	Returns the total number of command-line arguments, including the command.
PRECISION	Returns the number of significant digits for data of the same type as the argument.
PRESENT	Determines whether an optional argument is present.
RADIX	Returns the base for data of the same type as the argument.
RANGE	Returns the decimal exponent range for data of the same type as the argument.
SELECTED_INT_KIND	Returns the value of the kind parameter of integers in range *r*.
SELECTED_REAL_KIND	Returns the value of the kind parameter of reals with (optional) first argument digits and (optional) second argument exponent range. At least one optional argument is required.
SHAPE	Returns the shape of an array or scalar argument.
SIZEOF	Returns the number of bytes of storage used by the argument.
TINY	Returns the smallest positive number that can be represented by numbers of type the argument.
UBOUND	Returns the upper bounds for all dimensions of an array, or the upper bound for a specified dimension.

[1] i64 only

Random Number Intrinsic Procedures

The following table lists random number intrinsic procedures.

Name	Procedure Type	Description
RAN	Intrinsic function	Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1.
RANDOM_NUMBER	Intrinsic subroutine	Returns a pseudorandom real value greater than or equal to zero and less than one.
RANDOM_SEED	Intrinsic subroutine	Changes the starting point of RANDOM_NUMBER; takes one or no arguments.
RANDU	Intrinsic subroutine	Computes a pseudorandom number as a single-precision value.

The portability routines RANDOM and SEED also supply this functionality.

Date and Time Intrinsic Subroutines

The following table lists date and time intrinsic subroutines.

Name	Procedure Type	Description
CPU_TIME	Intrinsic subroutine	Returns the processor time in seconds.
DATE	Intrinsic subroutine	Returns the ASCII representation of the current date (in dd-mmm-yy form).
DATE_AND_TIME	Intrinsic subroutine	Returns the date and time. This is the preferred procedure for date and time.
IDATE	Intrinsic subroutine	Returns three integer values representing the current month, day, and year.

Name	Procedure Type	Description
SYSTEM_CLOCK	Intrinsic subroutine	Returns data from the system clock.
TIME	Intrinsic subroutine	Returns the ASCII representation of the current time (in hh:mm:ss form).

The portability routines GETDAT, GETTIM, SETDAT, and SETTIM also supply this functionality.

Keyboard and Speaker Library Routines

The following table lists keyboard and speaker library routines.

Name	Routine Type	Description
GETCHARQQ	Run-time Function	Returns the next keyboard keystroke.
BEEPQQ	Portability subroutine	Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.
GETSTRQQ	Run-time function	Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Run-time function	Checks the buffer to see if a keystroke is waiting.

Statements and Intrinsic Procedures for Memory Allocation and Deallocation

The following table lists statements and intrinsic procedures that are used for memory allocation and deallocation.

Name	Procedure Type	Description
ALLOCATE	Statement	Dynamically establishes allocatable array dimensions.

Name	Procedure Type	Description
ALLOCATED	Intrinsic Function	Determines whether an allocatable array is allocated.
DEALLOCATE	Statement	Frees the storage space previously reserved in an ALLOCATE statement.
FREE	Intrinsic Subroutine	Frees the memory block specified by the integer pointer argument.
MALLOC	Intrinsic Function	Allocates a memory block of size bytes and returns an integer pointer to the block.
MOVE_ALLOC	Intrinsic Subroutine	Moves an allocation from one allocatable object to another.

Intrinsic Functions for Arrays

The following table lists intrinsic functions for arrays.

Name	Description
ALL	Determines whether all array values meet the conditions in a mask along a (optional) dimension.
ANY	Determines whether any array values meet the conditions in a mask along a (optional) dimension.
COUNT	Counts the number of array elements that meet the conditions in a mask along a (optional) dimension.
CSHIFT	Performs a circular shift along a (optional) dimension.
DIMENSION	Identifies a variable as an array and specifies the number of elements.

Name	Description
DOT_PRODUCT	Performs dot-product multiplication on vectors (one-dimensional arrays).
EOSHIFT	Shifts elements off one end of *array* along a (optional) dimension and copies (optional) boundary values in other end.
LBOUND	Returns lower dimensional bounds of an array along a (optional) dimension.
MATMUL	Performs matrix multiplication on matrices (two-dimensional arrays).
MAXLOC	Returns the location of the maximum value in an array meeting conditions in a (optional) mask along a (optional) dimension.
MAXVAL	Returns the maximum value in an array along a (optional) dimension that meets conditions in a (optional) mask.
MERGE	Merges two arrays according to conditions in a mask.
MINLOC	Returns the location of the minimum value in an array meeting conditions in a (optional) mask along a (optional) dimension.
MINVAL	Returns the minimum value in an array along a (optional) dimension that meets conditions in a (optional) mask.
PACK	Packs an array into a vector (one-dimensional array) of an (optional) size using a mask.
PRODUCT	Returns product of elements of an array along a (optional) dimension that meet conditions in a (optional) mask.
RESHAPE	Reshapes an array with (optional) subscript order, padded with (optional) array elements.

Name	Description
SHAPE	Returns the shape of an array.
SIZE	Returns the extent of an array along a (optional) dimension.
SPREAD	Replicates an array by adding a dimension.
SUM	Sums array elements along a (optional) dimension that meet conditions of an (optional) mask.
TRANSPOSE	Transposes a two-dimensional array.
UBOUND	Returns upper dimensional bounds of an array along a (optional) dimension.
UNPACK	Unpacks a vector (one-dimensional array) into an array under a mask padding with values from a field.

The DIMENSION statement identifies variables as arrays.

Intrinsic Functions for Numeric and Type Conversion

The following table lists intrinsic functions for numeric and type conversion.

Name	Description
ABS	Returns the absolute value of the argument.
AIMAG	Returns imaginary part of complex number z.
AINT	Truncates the argument to a whole number of a specified (optional) kind.
AMAX0	Returns largest value among integer arguments as real.
AMIN0	Returns smallest value among integer arguments as real.

Name	Description
ANINT	Rounds to the nearest whole number of a specified (optional) kind.
CEILING	Returns smallest integer greater than the argument.
CMPLX	Converts the first argument and (optional) second argument to complex of a (optional) kind.
CONJG	Returns the conjugate of a complex number.
DBLE	Converts the argument to double precision type.
DCMPLX	Converts the argument to double complex type.
DFLOAT	Converts an integer to double precision type.
DIM	Returns the first argument - the second argument if positive; else 0.
DPROD	Returns double-precision product of two single precision arguments.
FLOAT	Converts the argument to REAL(4).
FLOOR	Returns the greatest integer less than or equal to the argument.
IFIX	Converts a single-precision real argument to an integer argument by truncating.
IMAG	Same as AIMAG.
INT	Converts a value to integer type.
LOGICAL	Converts between logical arguments of (optional) kind.
MAX	Returns largest value among arguments.

Name	Description
MAX1	Returns largest value among real arguments as integer.
MIN	Returns smallest value among arguments.
MIN1	Returns smallest value among real arguments as integer
MOD	Returns the remainder of the first argument divided by the second argument.
MODULO	Returns the first argument modulo of the second argument.
NINT	Returns the nearest integer to the argument.
REAL	Converts a value to real type.
SIGN	Returns absolute value of the first argument times the sign of the second argument.
SNGL	Converts a double-precision argument to single-precision real type.
TRANSFER	Transforms first argument into type of second argument with (optional) size if an array.
ZEXT	Extends the argument with zeros.

Trigonometric, Exponential, Root, and Logarithmic Intrinsic Procedures

The following table lists intrinsic procedures for trigonometric, exponential, root, and logarithmic operations.

Name	Description
ACOS	Returns the arccosine of the argument, expressed in radians between 0 and pi.
ACOSD	Returns the arccosine of the argument, expressed in degrees between 0 and 180.

Name	Description
ALOG	Returns natural log of the argument.
ALOG10	Returns common log (base 10) of the argument.
ASIN	Returns the arcsine of the argument, expressed in radians between ±pi/2.
ASIND	Returns the arcsine of the argument, expressed in degrees between ±90°.
ATAN	Returns the arctangent of the argument, expressed in radians between ±pi/2.
ATAND	Returns the arctangent of the argument, expressed in degrees between ±90°.
ATAN2	Returns the arctangent of the first argument divided by the second argument, expressed in radians between ±pi.
ATAN2D	Returns the arctangent of the first argument divided by the second argument, expressed in degrees between ±180°.
CCOS	Returns complex cosine of the argument.
CDCOS	Returns the double-precision complex cosine of the argument.
CDEXP	Returns double-precision complex exponential value of the argument.
CDLOG	Returns the double-precision complex natural log of the argument.
CDSIN	Returns the double-precision complex sine of the argument.
CDSQRT	Returns the double-precision complex square root of the argument.

Name	Description
CEXP	Returns the complex exponential value of the argument.
CLOG	Returns the complex natural log of the argument.
COS	Returns the cosine of the argument, which is in radians.
COSD	**COSD**(*x*). Returns the cosine of the argument, which is in degrees.
COSH	Returns the hyperbolic cosine of the argument.
COTAN	Returns the cotangent of the argument, which is in radians.
COTAND	Returns the cotangent of the argument, which is in degrees.
CSIN	Returns the complex sine of the argument.
CSQRT	Returns the complex square root of the argument.
DACOS	Returns the double-precision arccosine of the argument radians between 0 and pi.
DACOSD	Returns the arccosine of the argument in degrees between 0 and 180.
DASIN	Returns the double-precision arcsine of the argument in radians between ±pi/2.
DASIND	Returns the double-precision arcsine of the argument degrees between ±90°.
DATAN	Returns the double-precision arctangent of the argument radians between ±pi/2.

Name	Description
DATAND	Returns the double-precision arctangent of the argument degrees between ±90°.
DATAN2	Returns the double-precision arctangent of the second argument divided by the first argument radians between ±pi.
DATAN2D	Returns the double-precision arctangent of the second argument divided by the first argument degrees between ±180°.
DCOS	Returns the double-precision cosine of the argument radians.
DCOSD	Returns the double-precision cosine of the argument degrees.
DCOSH	Returns the double-precision hyperbolic cosine of the argument.
DCOTAN	Returns the double-precision cotangent of the argument.
DEXP	Returns the double-precision exponential value of the argument.
DLOG	Returns the double-precision natural log of the argument.
DLOG10	Returns the double-precision common log (base 10) of the argument.
DSIN	Returns the double-precision sin of the argument radians.
DSIND	Returns the double-precision sin of the argument degrees.
DSINH	Returns the double-precision hyperbolic sine of the argument.

Name	Description
DSQRT	Returns the double-precision square root of the argument.
DTAN	Returns the double-precision tangent of the argument radians.
DTAND	Returns the double-precision tangent of the argument degrees.
DTANH	Returns the double-precision hyperbolic tangent of the argument.
EXP	Returns the exponential value of the argument.
LOG	Returns the natural log of the argument.
LOG10	Returns the common log (base 10) of the argument.
SIN	Returns the sine of the argument, which is in radians.
SIND	Returns the sine of the argument, which is in degrees.
SINH	Returns the hyperbolic sine of the argument.
SQRT	Returns the square root of the argument.
TAN	Returns the tangent of the argument, which is in radians.
TAND	Returns the tangent of the argument, which is in degrees.
TANH	Returns the hyperbolic tangent of the argument.

Intrinsic Functions for Floating-Point Inquiry and Control

The following table lists intrinsic functions for floating-point inquiry and control.

Certain functions (EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT and SPACING) return values related to components of the model set of real numbers. For a description of this model, see the Model for Real Data.

Name	Description
DIGITS	Returns number of significant digits for data of the same type as the argument.
EPSILON	Returns the smallest positive number that when added to one produces a number greater than one for data of the same type as the argument.
EXPONENT	Returns the exponent part of the representation of x.
FRACTION	Returns the fractional part of the representation of the argument.
HUGE	Returns largest number that can be represented by data of type the argument.
MAXEXPONENT	Returns the largest positive decimal exponent for data of the same type as the argument.
MINEXPONENT	Returns the largest negative decimal exponent for data of the same type as the argument.
NEAREST	Returns the nearest different machine representable number to the first argument in the direction of the sign of the second argument.
PRECISION	Returns the number of significant digits for data of the same type as the argument.
RADIX	Returns the base for data of the same type as the argument.
RANGE	Returns the decimal exponent range for data of the same type as the argument.

Name	Description
RRSPACING	Returns the reciprocal of the relative spacing of numbers near the argument.
SCALE	Multiplies the first argument by 2 raised to the power of the second argument.
SET_EXPONENT	Returns a number whose fractional part is the first argument and whose exponential part is the second argument.
SPACING	Returns the absolute spacing of numbers near the argument.
TINY	Returns smallest positive number that can be represented by data of type of the argument.

The portability routines GETCONTROLFPQQ, GETSTATUSFPQQ, LCWRQQ, SCWRQQ, SETCONTROLFPQQ, and SSWRQQ also supply this functionality.

Character Intrinsic Functions

The following table lists character intrinsic functions.

Name	Description
ACHAR	Returns character in a specified position in the ASCII character set.
ADJUSTL	Adjusts left, removing leading blanks and inserting trailing blanks.
ADJUSTR	Adjusts right, removing trailing blanks and inserting leading blanks.
CHAR	Returns character in a specified position in the processor's character set of (optional) kind.
IACHAR	Returns the position of the argument in the ASCII character set.

Name	Description
ICHAR	Returns the position of the argument in the processor's character set.
INDEX	Returns the starting position of a substring in a string, leftmost or (optional) rightmost occurance.
LEN	Returns the size of the argument.
LEN_TRIM	Returns the number of characters in the argument, not counting trailing blanks.
LGE	Tests whether the the first argument is greater than or equal to the second argument, based on the ASCII collating sequence.
LGT	Tests whether the first argument is greater than the second argument, based on the ASCII collating sequence.
LLE	Tests whether the first argument is less than or equal to the second argument, based on the ASCII collating sequence.
LLT	Tests whether the first argument is less than the second argument, based on the ASCII collating sequence.
REPEAT	Concatenates multiple copies of a string.
SCAN	Scans a string for any characters in a set and returns leftmost or (optional) rightmost position where a match is found.
TRIM	Removes trailing blanks from a string.
VERIFY	Returns the position of the leftmost or (optional) rightmost character in the argument string not in a set, or zero if all characters in the set are present.

Intrinsic Procedures for Bit Operation and Representation

The following tables list intrinsic procedures for bit operation and representation.

Table 711: Bit Operation

Name	Procedure Type	Description
BIT_SIZE	Intrinsic Function	Returns the number of bits in integers of type the argument.
BTEST	Intrinsic Function	Tests a bit in a position of the argument; true if bit is 1.
IAND	Intrinsic Function	Performs a logical AND.
IBCHNG	Intrinsic Function	Reverses value of bit in a position of the argument.
IBCLR	Intrinsic Function	Clears the bit in a position of the argument to zero.
IBITS	Intrinsic Function	Extracts a sequence of bits of length from the argument starting in a position.
IBSET	Intrinsic Function	Sets the bit in a position of the argument to one.
IEOR	Intrinsic Function	Performs an exclusive OR.
IOR	Intrinsic Function	Performs an inclusive OR.

Name	Procedure Type	Description
ISHA	Intrinsic Function	Shifts the argument arithmetically left or right by shift bits; left if shift positive, right if shift negative. Zeros shifted in from the right, ones shifted in from the left.
ISHC	Intrinsic Function	Performs a circular shift of the argument left or right by shift bits; left if shift positive, right if shift negative. No bits lost.
ISHFT	Intrinsic Function	Shifts the argument logically left or right by shift bits; left if shift positive, right if shift negative. Zeros shifted in from opposite end.
ISHFTC	Intrinsic Function	Performs a circular shift of the rightmost bits of (optional) size by shift bits. No bits lost.
ISHL	Intrinsic Function	Shifts the argument logically left or right by shift bits. Zeros shifted in from opposite end.
MVBITS	Intrinsic Subroutine	Copies a sequence of bits from one integer to another.

Name	Procedure Type	Description
NOT	Intrinsic Function	Performs a logical complement.

Table 712: Bit Representation

Name	Procedure Type	Description
LEADZ	Intrinsic Function	Returns leading zero bits in an integer.
POPCNT	Intrinsic Function	Returns number of 1 bits in an integer.
POPPAR	Intrinsic Function	Returns the parity of an integer.
TRAILZ	Intrinsic Function	Returns trailing zero bits in an integer.

QuickWin Library Routines (W*32, W*64)

The following table lists Quickwin library routines.

Programs that use these routines must access the appropriate library with USE IFQWIN. These routines are restricted to Windows* systems.

Name	Routine Type	Description
ABOUTBOXQQ	Function	Adds an About Box with customized text.
APPENDMENUQQ	Function	Appends a menu item.
CLICKMENUQQ	Function	Sends menu click messages to the application window.
DELETEMENUQQ	Function	Deletes a menu item.
FOCUSQQ	Function	Makes a child window active, and gives focus to the child window.

Name	Routine Type	Description
GETACTIVEQQ	Function	Gets the unit number of the active child window.
GETEXITQQ	Function	Gets the setting for a QuickWin application's exit behavior.
GETHWNDQQ	Function	Gets the true windows handle from window with the specified unit number.
GETWINDOWCONFIG	Function	Returns the current window's properties.
GETWSIZEQQ	Function	Gets the size of the child or frame window.
GETUNITQQ	Function	Gets the unit number corresponding to the specified windows handle. Inverse of GETHWNDQQ.
INCHARQQ	Function	Reads a keyboard input and return its ASCII value.
INITIALSETTINGS	Function	Controls initial menu settings and initial frame window settings.
INQFOCUSQQ	Function	Determines which window is active and has the focus.
INSERTMENUQQ	Function	Inserts a menu item.
INTEGERTORGB	Subroutine	Converts a true color value into its red, green and blue components.
MESSAGEBOXQQ	Function	Displays a message box.
MODIFYMENUFLAGSQQ	Function	Modifies a menu item state.

Name	Routine Type	Description
MODIFYMENUROUTINEQQ	Function	Modifies a menu item's callback routine.
MODIFYMENUSTRINGQQ	Function	Changes a menu item's text string.
PASSDIRKEYSQQ	Function	Determines the behavior of direction and page keys.
REGISTERMOUSEEVENT	Function	Registers the application-defined routines to be called on mouse events.
RGBTOINTEGER	Function	Converts a trio of red, green and blue values to a true color value for use with RGB functions and subroutines.
SETACTIVEQQ	Function	Makes the specified window the current active window without giving it the focus.
SETEXITQQ	Function	Sets a QuickWin application's exit behavior.
SETMESSAGEQQ	Subroutine	Changes any QuickWin message, including status bar messages, state messages and dialog box messages.
SETMOUSECURSOR	Function	Sets the mouse cursor for the window in focus.
SETWINDOWCONFIG	Function	Configures the current window's properties.
SETWINDOWMENUQQ	Function	Sets the Window menu to which current child window names will be appended.
SETWSIZEQQ	Function	Sets the size of the child or frame window.

Name	Routine Type	Description
UNREGISTERMOUSEEVENT	Function	Removes the callback routine registered by REGISTERMOUSEEVENT.
WAITONMOUSEEVENT	Function	Blocks return until a mouse event occurs.

Graphics Library Routines (W*32, W*64)

The following table lists library routines for graphics.

Programs that use these routines must access the appropriate library with USE IFQWIN. These routines are restricted to Windows* systems.

Name	Routine Type	Description
ARC, ARC_W	Functions	Draws an arc.
CLEARSCREEN	Subroutine	Clears the screen, viewport, or text window.
DISPLAYCURSOR	Function	Turns the cursor off and on.
ELLIPSE, ELLIPSE_W	Functions	Draws an ellipse or circle.
FLOODFILL, FLOODFILL_W	Functions	Fills an enclosed area of the screen with the current color index, using the current fill mask.
FLOODFILLRGB, FLOODFILLRGB_W	Functions	Fills an enclosed area of the screen with the current RGB color, using the current fill mask.
GETARCINFO	Function	Determines the end points of the most recently drawn arc or pie.
GETBKCOLOR	Function	Returns the current background color index.

Name	Routine Type	Description
GETBKCOLORRGB	Function	Returns the current background RGB color.
GETCOLOR	Function	Returns the current color index.
GETCOLORRGB	Function	Returns the current RGB color.
GETCURRENTPOSITION, GETCURRENTPOSITION_W	Subroutines	Returns the coordinates of the current graphics-output position.
GETFILLMASK	Subroutine	Returns the current fill mask.
GETFONTINFO	Function	Returns the current font characteristics.
GETGTEXTEXTENT	Function	Determines the width of the specified text in the current font.
GETGTEXTROTATION	Function	Get the current text rotation angle.
GETIMAGE, GETIMAGE_W	Subroutines	Stores a screen image in memory.
GETLINESTYLE	Function	Returns the current line style.
GETPHYSCOORD	Subroutine	Converts viewport coordinates to physical coordinates.
GETPIXEL, GETPIXEL_W	Functions	Returns a pixel's color index.
GETPIXELRGB, GETPIXELRGB_W	Functions	Returns a pixel's RGB color.
GETPIXELS	Function	Returns the color indices of multiple pixels.

Name	Routine Type	Description
GETPIXELSRGB	Function	Returns the RGB colors of multiple pixels.
GETTEXTCOLOR	Function	Returns the current text color index.
GETTEXTCOLORRGB	Function	Returns the current text RGB color.
GETTEXTPOSITION	Subroutine	Returns the current text-output position.
GETTEXTWINDOW	Subroutine	Returns the boundaries of the current text window.
GETVIEWCOORD, GETVIEWCOORD_W	Subroutines	Converts physical or window coordinates to viewport coordinates.
GETWINDOWCOORD	Subroutine	Converts viewport coordinates to window coordinates.
GETWRITEMODE	Function	Returns the logical write mode for lines.
GRSTATUS	Function	Returns the status (success or failure) of the most recently called graphics routine.
IMAGESIZE, IMAGESIZE_W	Functions	Returns image size in bytes.
INITIALIZEFONTS	Function	Initializes the font library.
LINETO, LINETO_W	Functions	Draws a line from the current position to a specified point.
LINETOAR	Function	Draws a line between points in one array and corresponding points in another array.

Name	Routine Type	Description
LINETOAREX	Function	Similar to LINETOAR, but also lets you specify color and line style.
LOADIMAGE, LOADIMAGE_W	Functions	Reads a Windows bitmap file (.BMP) and displays it at the specified location.
MOVETO, MOVETO_W	Subroutines	Moves the current position to the specified point.
OUTGTEXT	Subroutine	Sends text in the current font to the screen at the current position.
OUTTEXT	Subroutine	Sends text to the screen at the current position.
PIE, PIE_W	Functions	Draws a pie slice.
POLYBEZIER, POLYBEZIER_W	Functions	Draws one or more Bezier curves.
POLYBEZIERTO, POLYBEZIERTO_W	Functions	Draws one or more Bezier curves.
POLYGON, POLYGON_W	Functions	Draws a polygon.
POLYLINEQQ	Function	Draws a line between successive points in an array.
PUTIMAGE, PUTIMAGE_W	Subroutines	Retrieves an image from memory and displays it.
RECTANGLE, RECTANGLE_W	Functions	Draws a rectangle.
REMAPALLPALETTERGB	Function	Remaps a set of RGB color values to indices recognized by the current video configuration.

Name	Routine Type	Description
REMAPPALETTERGB	Function	Remaps a single RGB color value to a color index.
SAVEIMAGE, SAVEIMAGE_W	Functions	Captures a screen image and saves it as a Windows bitmap file.
SCROLLTEXTWINDOW	Subroutine	Scrolls the contents of a text window.
SETBKCOLOR	Function	Sets the current background color.
SETBKCOLORRGB	Function	Sets the current background color to a direct color value rather than an index to a defined palette.
SETCLIPRGN	Subroutine	Limits graphics output to a part of the screen.
SETCOLOR	Function	Sets the current color to a new color index.
SETCOLORRGB	Function	Sets the current color to a direct color value rather than an index to a defined palette.
SETFILLMASK	Subroutine	Changes the current fill mask to a new pattern.
SETFONT	Function	Finds a single font matching the specified characteristics and assigns it to OUTGTEXT.
SETGTEXTROTATION	Subroutine	Sets the direction in which text is written to the specified angle.
SETLINESTYLE	Subroutine	Changes the current line style.

Name	Routine Type	Description
SETPIXEL, SETPIXEL_W	Functions	Sets color of a pixel at a specified location.
SETPIXELRGB, SETPIXELRGB_W	Functions	Sets RGB color of a pixel at a specified location.
SETPIXELS	Subroutine	Sets the color indices of multiple pixels.
SETPIXELSRGB	Subroutine	Sets the RGB color of multiple pixels.
SETTEXTCOLOR	Function	Sets the current text color to a new color index.
SETTEXTCOLORRGB	Function	Sets the current text color to a direct color value rather than an index to a defined palette.
SETTEXTCURSOR	Function	Sets the height and width of the text cursor for the window in focus.
SETTEXTPOSITION	Subroutine	Changes the current text position.
SETTEXTWINDOW	Subroutine	Sets the current text display window.
SETVIEWORG	Subroutine	Positions the viewport coordinate origin.
SETVIEWPORT	Subroutine	Defines the size and screen position of the viewport.
SETWINDOW	Function	Defines the window coordinate system.
SETWRITEMODE	Function	Changes the current logical write mode for lines.

Name	Routine Type	Description
WRAPON	Function	Turns line wrapping on or off.

Portability Library Routines

The following tables list library routines for portability.

Programs that use these routines must access the portability library with USE IFPORT.

Table 715: Information Retrieval

Name	Procedure Type	Description
FSTAT	Function	Returns information about a logical file unit.
GETCWD	Function	Returns the pathname of the current working directory.
GETENV	Function	Searches the environment for a given string and returns its value if found.
GETGID	Function	Returns the group ID of the user.
GETLOG	Subroutine	Returns the user's login name.
GETPID	Function	Returns the process ID of the process.
GETUID	Function	Returns the user ID of the user of the process.
HOSTNAM [1]	Function	Returns the name of the user's host.
ISATTY	Function	Checks whether a logical unit number is a terminal.

Name	Procedure Type	Description
LSTAT	Function	Returns information about a named file. STAT is the preferred form of this function.
RENAME	Function	Renames a file.
STAT	Function	Returns information about a named file.
UNLINK	Function	Deletes the file given by path.

Table 716: Process Control

Name	Procedure Type	Description
ABORT	Subroutine	Stops execution of the current process, clears I/O buffers, and writes a string to external unit 0.
ALARM	Function	Executes an external subroutine after waiting a specified number of seconds.
KILL	Function	Sends a signal code to the process given by ID.
SIGNAL	Function	Changes the action for signal.
SLEEP	Subroutine	Suspends program execution for a specified number of seconds.
SYSTEM	Function	Executes a command in a separate shell.

Table 717: Numeric Values and Conversion

Name	Procedure Type	Description
BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN	Functions	Return single-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
BIC, BIS, BIT	Subroutines Function	Perform bit level clear, set, and test for integers.
CDFLOAT	Function	Converts a COMPLEX(4) argument to DOUBLE PRECISION type.
COMPLINT, COMPLREAL, COMPLLOG	Functions	Return a BIT-WISE complement or logical .NOT. of the argument.
CSMG	Function	Performs an effective BIT-WISE store under mask.
DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN	Functions	Return double-precision values of Bessel functions of the first and second kind of orders 1, 2, and n, respectively.
DFLOATI, DFLOATJ, DFLOATK	Functions	Convert an integer to double-precision real type.
DRAND, DRANDM	Functions	Return random numbers between 0.0 and 1.0.
DRANSET	Subroutine	Sets the seed for the random number generator.
IDFLOAT	Function	Converts an INTEGER(4) argument to double-precision real type.
IFLOATI, IFLOATJ	Functions	Convert an integer to single-precision real type.

Name	Procedure Type	Description
INMAX	Function	Returns the maximum positive value for an integer.
INTC	Function	Converts an INTEGER(4) argument to INTEGER(2) type.
IRAND, IRANDM	Functions	Return a positive integer in the range 0 through 2**31-1 or 2**15-1 if called without an argument.
IRANGET	Subroutine	Returns the current seed.
IRANSET	Subroutine	Sets the seed for the random number generator.
JABS	Function	Computes an absolute value.
LONG	Function	Converts an INTEGER(2) argument to INTEGER(4) type.
QRANSET	Subroutine	Sets the seed for a sequence of pseudo-random numbers.
RAND, RANDOM [2]	Functions	Return random values in the range 0 through 1.0.
RANF	Function	Generates a random number between 0.0 and RAND_MAX.
RANGET	Subroutine	Returns the current seed.
RANSET	Subroutine	Sets the seed for the random number generator.
SEED	Subroutine	Changes the starting point of RANDOM.

Name	Procedure Type	Description
SHORT	Function	Converts an INTEGER(4) argument to INTEGER(2) type.
SRAND	Subroutine	Seeds the random number generator used with IRAND and RAND.

Table 718: Input and Output

Name	Procedure Type	Description
ACCESS	Function	Checks a file for accessibility according to mode.
CHMOD	Function	Changes file attributes.
FGETC	Function	Reads a character from an external unit.
FLUSH	Subroutine	Flushes the buffer for an external unit to its associated file.
FPUTC	Function	Writes a character to an external unit.
FSEEK	Subroutine	Repositions a file on an external unit.
FTELL, FTELLI8	Function	Return the offset, in bytes, from the beginning of the file.
GETC	Function	Reads a character from unit 5.
GETPOS, GETPOSI8	Functions	Return the offset, in bytes, from the beginning of the file.
PUTC	Function	Writes a character to unit 6.

Table 719: Date and Time

Name	Procedure Type	Description
CLOCK	Function	Returns current time in HH:MM:SS format using a 24-hour clock.
CLOCKX	Subroutine	Returns the processor clock to the nearest microsecond.
CTIME	Function	Converts system time to a 24-character ASCII string.
DATE [3]	Subroutine or Function	Returns the current system date.
DATE4	Subroutine	Returns the current system date.
DCLOCK	Function	Returns the elapsed time in seconds since the start of the current process.
DTIME	Function	Returns CPU time since later of (1) start of program, or (2) most recent call to DTIME.
ETIME	Function	Returns elapsed CPU time since the start of program execution.
FDATE	Subroutine or Function	Returns the current date and time as an ASCII string.
GETDAT	Subroutine	Returns the date.
GETTIM	Subroutine	Returns the time.
GMTIME	Subroutine	Returns Greenwich Mean Time as a 9-element integer array.

Name	Procedure Type	Description
IDATE [3]	Subroutine	Returns the date either as one 3-element array or three scalar parameters (month, day, year).
IDATE4	Subroutine	Returns the date either as one 3-element array or three scalar parameters (month, day, year).
ITIME	Subroutine	Returns current time as a 3-element array (hour, minute, second).
JDATE [3]	Function	Returns current date as an 8-character string with the Julian date.
JDATE4	Function	Returns current date as a 10-character string with the Julian date.
LTIME	Subroutine	Returns local time as a 9-element integer array.
RTC	Function	Returns number of seconds since 00:00:00 GMT, Jan 1, 1970.
SECNDS	Function	Returns number of seconds since midnight, less the value of its argument.
SETDAT	Function	Sets the date.
SETTIM	Function	Sets the time.
TIME	Subroutine or Function	As a subroutine, returns time formatted as HH:MM:SS; as a function, returns time in seconds since 00:00:00 GMT, Jan 1, 1970.

Name	Procedure Type	Description
TIMEF	Function	Returns the number of seconds since the first time this function was called (or zero).

Table 720: Error Handling

Name	Procedure Type	Description
GETLASTERROR	Function	Returns the last error set.
GETLASTERRORQQ	Function	Returns the last error set by a run-time function or subroutine.
IERRNO	Function	Returns the last code error.
SETERRORMODEQQ	Subroutine	Sets the mode for handling critical errors.

Table 721: Program Control

Name	Procedure Type	Description
RAISEQQ	Function	Sends an interrupt to the executing program, simulating an interrupt from the operating system.
RUNQQ	Function	Calls another program and waits for it to execute.
SIGNALQQ	Function	Controls signal handling.
SLEEPQQ	Subroutine	Delays execution of the program for the specified time.

Table 722: System, Drive, and Directory

Name	Procedure Type	Description
CHDIR	Function	Changes the current working directory.
CHANGEDIRQQ	Function	Makes the specified directory the current (default) directory.
CHANGEDRIVEQQ	Function	Makes the specified drive the current drive.
DELDIRQQ	Function	Deletes a specified directory.
GETDRIVEDIRQQ	Function	Returns the current drive and directory path.
GETDRIVESIZEQQ	Function	Gets the size of the specified drive.
GETDRIVESQQ	Function	Reports the drives available to the system.
GETENVQQ	Function	Gets a value from the current environment.
MAKEDIRQQ	Function	Makes a directory with the specified directory name.
SETENVQQ	Function	Adds a new environment variable, or sets the value of an existing one.
SYSTEMQQ	Function	Executes a command by passing a command string to the operating system's command interpretor.

Table 723: Speaker

Name	Procedure Type	Description
BEEPQQ	Subroutine	Sounds the speaker for a specified duration in milliseconds at a specified frequency in Hertz.

Table 724: File Management

Name	Procedure Type	Description
DELFILESQQ	Function	Deletes the specified files in a specified directory.
FINDFILEQQ	Function	Searches for a file in the directories specified in the PATH environment variable.
FULLPATHQQ	Function	Returns the full path for a specified file or directory.
GETFILEINFOQQ	Function	Returns information about files with names that match a request string.
PACKTIMEQQ	Subroutine	Packs time values for use by SETFILETIMEQQ.
RENAMEFILEQQ	Function	Renames a file.
SETFILEACCESSQQ	Function	Sets file-access mode for the specified file.
SETFILETIMEQQ	Function	Sets modification time for a given file.
SPLITPATHQQ	Function	Breaks a full path into four components.
UNPACKTIMEQQ	Subroutine	Unpacks a file's packed time and date value into its component parts.

Table 725: Arrays

Name	Procedure Type	Description
BSEARCHQQ	Function	Performs a binary search for a specified element on a sorted one-dimensional array of non-structure data types (derived types are not allowed).
SORTQQ	Subroutine	Sorts a one-dimensional array of non-structure data types (derived types are not allowed).

Table 726: Floating-Point Inquiry and Control

Name	Procedure Type	Description
CLEARSTATUSFPQQ	Subroutine	Clears the exception flags in the floating-point processor status word.
GETCONTROLFPQQ	Subroutine	Returns the value of the floating-point processor control word.
GETSTATUSFPQQ	Subroutine	Returns the value of the floating-point processor status word.
LCWRQQ	Subroutine	Same as SETCONTROLFPQQ.
SCWRQQ	Subroutine	Same as GETCONTROLFPQQ.
SETCONTROLFPQQ	Subroutine	Sets the value of the floating-point processor control word.
SSWRQQ	Subroutine	Same as GETSTATUSFPQQ.

Table 727: IEEE Functionality

Name	Procedure Type	Description
IEEE_FLAGS	Function	Sets, gets, or clears IEEE flags.
IEEE_HANDLER	Function	Establishes a handler for IEEE exceptions.

Table 728: Serial Port I/O[4]

Name	Procedure Type	Description
SPORT_CANCEL_IO	Function	Cancels any I/O in progress to the specified port.
SPORT_CONNECT	Function	Establishes the connection to a serial port and defines certain usage parameters.
SPORT_CONNECT_EX	Function	Establishes the connection to a serial port, defines certain usage parameters, and defines the size of the internal buffer for data reception.
SPORT_GET_HANDLE	Function	Returns the Windows* handle associated with the communications port.
SPORT_GET_STATE	Function	Returns the baud rate, parity, data bits, and stop bit settings of the communications port.
SPORT_GET_STATE_EX	Function	Returns the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.
SPORT_GET_TIMEOUTS	Function	Returns the user selectable timeouts for the serial port.

Name	Procedure Type	Description
SPORT_PEEK_DATA	Function	Returns information about the availability of input data.
SPORT_PEEK_LINE	Function	Returns information about the availability of input records.
SPORT_PURGE	Function	Executes a purge function on the specified port.
SPORT_READ_DATA	Function	Reads available data from the port specified.
SPORT_READ_LINE	Function	Reads a record from the port specified.
SPORT_RELEASE	Function	Releases a serial port that has previously been connected.
SPORT_SET_STATE	Function	Sets the baud rate, parity, data bits and stop bit settings of the communications port.
SPORT_SET_STATE_EX	Function	Sets the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.
SPORT_SET_TIMEOUTS	Function	Sets the user selectable timeouts for the serial port.
SPORT_SHOW_STATE	Function	Displays the state of a port.
SPORT_SPECIAL_FUNC	Function	Executes a communications function on a specified port.
SPORT_WRITE_DATA	Function	Outputs data to a specified port.

Name	Procedure Type	Description
SPORT_WRITE_LINE	Function	Outputs data to a specified port and follows it with a record terminator.

Table 729: Miscellaneous

Name	Procedure Type	Description
LNBLNK	Function	Returns the index of the last non-blank character in a string.
QSORT	Subroutine	Returns a sorted version of a one-dimensional array of a specified number of elements of a named size.
RINDEX	Function	Returns the index of the last occurrence of a substring in a string.
SCANENV	Subroutine	Scans the environment for the value of an environment variable.
TTYNAM	Function	Checks whether a logical unit is a terminal.

[1] This routine can also be specified as HOSTNM.

[2] There is also a RANDOM subroutine in the portability library.

[3] The two-digit year return value of DATE, IDATE, and JDATE may cause problems with the year 2000. Use the intrinsic subroutine DATE_AND_TIME instead.

[4] W*32, W*64

See Also

- Language Summary Tables

National Language Support Library Routines (W*32, W*64)

The following table lists library routines for National Language Support (NLS).

Programs that use these routines must access the NLS library with USE IFNLS. These routines are restricted to Windows* systems.

Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Name	Routine Type	Description
MBCharLen	Function	Returns the length of the first multibyte character in a string.
MBConvertMBToUnicode	Function	Converts a character string from a multibyte codepage to a Unicode string.
MBConvertUnicodeToMB	Function	Converts a Unicode string to a multibyte character string of the current codepage.
MBCurMax	Function	Returns the longest possible mutlibyte character for the current codepage.
MBINCHARQQ	Function	Same as INCHARQQ, but can read a single multibyte character at once.
MBINDEX	Function	Same as INDEX, except that multibyte characters can be included in its arguments.
MBJISToJMS	Function	Converts a Japan Industry Standard (JIS) character to a Kanji (Shift JIS or JMS) character.
MBJMSToJIS	Function	Converts a Kanji (Shift JIS or JMS) character to a Japan Industry Standard (JIS) character.
MBLead	Function	Determines whether a given character is the first byte of a multibyte character.

Name	Routine Type	Description
MBLen	Function	Returns the number of multibyte characters in a string, including trailing spaces.
MBLen_Trim	Function	Returns the number of multibyte characters in a string, not including trailing spaces.
MBLGE, MBLGT, MBLLE, MBLLT,MBLEQ, MBLNE	Function	Same as LGE, LGT, LLE, and LLT, and the logical operators .EQ. and .NE., except that multibyte characters can be included in their arguments.
MBNext	Function	Returns the string position of the first byte of the multibyte character immediately after the given string position.
MBPrev	Function	Returns the string position of the first byte of the multibyte character immediately before the given string position.
MBSCAN	Function	Same as SCAN, except that multibyte characters can be included. in its arguments
MBStrLead	Function	Performs a context sensitive test to determine whether a given byte in a character string is a lead byte.
MBVERIFY	Function	Same as VERIFY, except that multibyte characters can be included in its arguments.
NLSEnumCodepages	Function	Returns an array of valid codepages for the current console.

Name	Routine Type	Description
NLSEnumLocales	Function	Returns an array of locales (langauge/country combinations) installed on the system.
NLSFormatCurrency	Function	Formats a currency number according to conventions of the current locale (language/country).
NLSFormatDate	Function	Formats a date according to conventions of the current locale (language/country).
NLSFormatNumber	Function	Formats a number according to conventions of the current locale (language/country).
NLSFormatTime	Function	Formats a time according to conventions of the current locale (language/country).
NLSGetEnvironmentCodepage	Function	Returns the current codepage for the system Window or console.
NLSGetLocale	Subroutine	Returns the current language, country, and/or codepage.
NLSGetLocaleInfo	Function	Returns information about the current locale.
NLSSetEnvironmentCodepage	Function	Sets the codepage for the console.
NLSSetLocale	Function	Sets the current language, country, and codepage.

POSIX* Library Procedures

The following table lists library procedures for POSIX*.

Programs that use POSIX procedures must access the appropriate libraries with USE IFPOSIX. The IPX *nnnn* routines are functions; the PXF *nnnn* routines are subroutines, except for the routines named PXFIS *nnnn* and PXFWIF *nnnn*.

Name	Description
IPXFARGC	Returns the index of the last commans-line argument.
IPXFCONST	Returns the value associated with a constant defined in the C POSIX standard.
IPXFLENTRIM	Returns the index of the last non-blank character in an input string.
IPXFWEXITSTATUS[1]	Returns the exit code of a child process.
IPXFWSTOPSIG [1]	Returns the number of the signal that caused a child process to stop.
IPXFWTERMSIG [1]	Returns the number of the signal that caused a child process to terminate.
PXF(type)GET	Gets the value stored in a component (or field) of a structure.
PXF(type)SET	Sets the value of a component (or field) of a structure.
PXFA(type)GET	Gets the array values stored in a component (or field) of a structure.
PXFA(type)SET	Sets the value of an array component (or field) of a structure.
PXFACCESS	Determines the accessibility of a file.
PXFALARM	Schedules an alarm.
PXFCALLSUBHANDLE	Calls the associated subroutine.
PXFCFGETISPEED [1]	Returns the input baud rate from a `termios` structure.

Name	Description
PXFCFGETOSPEED [1]	Returns the output baud rate from a `termios` structure.
PXFCFSETISPEED [1]	Sets the input baud rate in a `termios` structure.
PXFCFSETOSPEED [1]	Sets the output baud rate in a `termios` structure.
PXFCHDIR	Changes the current working directory.
PXFCHMOD	Changes the ownership mode of the file.
PXFCHOWN [1]	Changes the owner and group of a file.
PXFCLEARENV	Clears the process environment.
PXFCLOSE	Closes the file associated with the descriptor.
PXFCLOSEDIR	Closes the directory stream.
PXFCNTL [1]	Manipulates an open file descriptor.
PXFCONST	Returns the value associated with a constant.
PXFCREAT	Creates a new file or rewrites an existing file.
PXFCTERMID [1]	Generates a terminal pathname.
PXFDUP, PXFDUP2	Duplicates an existing file descriptor.
PXFE(type)GET	Gets the value stored in an array element component (or field) of a structure.
PXFE(type)SET	Sets the value of an array element component (or field) of a structure.
PXFEXECV, PXFEXECVE, PXFEXECVP	Executes a new process by passing command-line arguments.
PXFEXIT, PXFFASTEXIT	Exits from a process.

2306

Name	Description
PXFFDOPEN	Opens an external unit.
PXFFFLUSH	Flushes a file directly to disk.
PXFFGETC	Reads a character from a file.
PXFFILENO	Returns the file descriptor associated with a specified unit.
PXFFORK [1]	Creates a child process that differs from the parent process only in its PID.
PXFFPATHCONF	Gets the value for a configuration option of an opened file.
PXFFPUTC	Writes a character to a file.
PXFFSEEK	Modifies a file position.
PXFFSTAT	Gets a file's status information.
PXFFTELL	Returns the relative position in bytes from the beginning of the file.
PXFGETARG	Gets the specified command-line argument.
PXFGETATTY	Tests whether a file descriptor is connected to a terminal.
PXFGETC	Reads a character from standard input unit 5.
PXFGETCWD	Returns the path of the current working directory.
PXFGETEGID [1]	Gets the effective group ID of the current process.
PXFGETENV	Gets the setting of an environment variable.
PXFGETEUID [1]	Gets the effective user ID of the current process.

Name	Description
PXFGETGID [1]	Gets the real group ID of the current process.
PXFGETGRGID [1]	Gets group information for the specified GID.
PXFGETGRNAM [1]	Gets group information for the named group.
PXFGETGROUPS [1]	Gets supplementary group IDs.
PXFGETLOGIN	Gets the name of the user.
PXFGETPGRP [1]	Gets the process group ID of the calling process.
PXFGETPID	Gets the process ID of the calling process.
PXFGETPPID	Gets the process ID of the parent of the calling process.
PXFGETPWNAM [1]	Gets password information for a specified name.
PXFGETPWUID [1]	Gets password information for a specified UID.
PXFGETSUBHANDLE	Returns a subroutine handle for a subroutine.
PXFGETUID [1]	Gets the real user ID of the current process.
PXFISBLK	Tests for a block special file.
PXFISCHR	Tests for a character file.
PXFISCONST	Tests whether a string is a valid constant name.
PXFISDIR	Tests whether a file is a directory.
PXFISFIFO	Tests whether a file is a special FIFO file.
PXFISREG	Tests whether a file is a regular file.
PXFKILL	Sends a signal to a specified process.

2308

Name	Description
PXFLINK	Creates a link to a file or directory.
PXFLOCALTIME	Converts a given elapsed time in seconds to local time.
PXFLSEEK	Positions a file a specified distance in bytes.
PXFMKDIR	Creates a new directory.
PXFMKFIFO [1]	Creates a new FIFO.
PXFOPEN	Opens or creates a file.
PXFOPENDIR	Opens a directory and associates a stream with it.
PXFPATHCONF	Gets the value for a configuration option of an opened file.
PXFPAUSE	Suspends process execution.
PXFPIPE	Creates a communications pipe between two processes.
PXFPOSIXIO [1]	Sets the current value of the POSIX I/O flag.
PXFPUTC	Outputs a character to logical unit 6 (stdout).
PXFREAD	Reads from a file.
PXFREADDIR	Reads the current directory entry.
PXFRENAME	Changes the name of a file.
PXFREWINDDIR	Resets the position of the stream to the beginning of the directory.
PXFRMDIR	Removes a directory.
PXFSETENV	Adds a new environment variable or sets the value of an environment variable.

Name	Description
PXFSETGID [1]	Sets the effective group ID of the current process.
PXFSETPGID [1]	Sets the process group ID.
PXFSETSID [1]	Creates a session and sets the process group ID.
PXFSETUID [1]	Sets the effective user ID of the current process.
PXFSIGACTION	Changes the action associated with a specific signal.
PXFSIGADDSET [1]	Adds a signal to a signal set.
PXFSIGDELSET [1]	Deletes a signal from a signal set.
PXFSIGEMPTYSET [1]	Empties a signal set.
PXFSIGFILLSET [1]	Fills a signal set.
PXFSIGISMEMBER	Tests whether a signal is a member of a signal set.
PXFSIGPENDING [1]	Examines pending signals.
PXFSIGPROCMASK [1]	Changes the list of currently blocked signals.
PXFSIGSUSPEND [1]	Suspends the process until a signal is received.
PXFSLEEP	Forces the process to sleep.
PXFSTAT	Gets a file's status information.
PXFSTRUCTCOPY	Copies the contents of one structure to another.
PXFSTRUCTCREATE	Creates an instance of the specified structure.
PXFSTRUCTFREE	Deletes the instance of a structure.

[1] 2310

Name	Description
PXFSYSCONF	Gets values for system limits or options.
PXFTCDRAIN [1]	Waits until all output written has been transmitted.
PXFTCFLOW [1]	Suspends the transmission or reception of data.
PXFTCFLUSH [1]	Discards terminal input data, output data, or both.
PXFTCGETATTR [1]	Reads current terminal settings.
PXFTCGETPGRP [1]	Gets the foreground process group ID associated with the terminal.
PXFTCSENDBREAK [1]	Sends a break to the terminal.
PXFTCSETATTR [1]	Writes new terminal settings.
PXFTCSETPGRP [1]	Sets the foreground process group associated with the terminal.
PXFTIME	Gets the system time.
PXFTIMES	Gets process times.
PXFTTYNAM [1]	Gets the terminal pathname.
PXFUCOMPARE	Compares two unsigned integers.
PXFUMASK	Sets a new file creation mask and gets the previous one.
PXFUNAME	Gets the operation system name.
PXFUNLINK	Removes a directory entry.
PXFUTIME	Sets file access and modification times.
PXFWAIT [1]	Waits for a child process.
PXFWAITPID [1]	Waits for a specific PID.

Name	Description
PXFWIFEXITED [1]	Determines if a child process has exited.
PXFWIFSIGNALED [1]	Determines if a child process has exited because of a signal.
PXFWIFSTOPPED [1]	Determines if a child process has stopped.
PXFWRITE	Writes to a file.

[1] L*X, M*X

Dialog Library Routines (W*32, W*64)

The following table lists routines from the dialog library.

Programs that use these routines must access the Dialog library with USE IFLOGM. These routines are restricted to Windows* systems.

Name	Routine Type	Description
DLGEXIT	Subroutine	Closes an open dialog.
DLGFLUSH	Subroutine	Updates the display of a dialog box.
DLGGET	Function	Retrieves values of dialog control variables.
DLGGETCHAR	Function	Retrieves values of dialog control variables of type Character.
DLGGETINT	Function	Retrieves values of dialog control variables of type Integer.
DLGGETLOG	Function	Retrieves values of dialog control variables of type Logical.
DLGINIT	Function	Initializes a dialog.

Name	Routine Type	Description
DLGINITWITHRESOURCEHANDLE	Function	Initializes a dialog.
DLGISDLGMESSAGE	Function	Determines whether a message is intended for a modeless dialog box and, if it is, processes it.
DLGISDLGMESSAGEWITHDLG	Function	Determines whether a message is intended for a specific modeless dialog box and, if it is, processes it.
DLGMODAL	Function	Displays a dialog and processes dialog selections from user.
DLGMODALWITHPARENT	Function	Displays a dialog in a specific parent window and processes dialog selections from user.
DLGMODELESS	Function	Displays a modeless dialog box.
DLGSENDCTRLMESSAGE	Function	Sends a message to a dialog box control.
DLGSET	Function	Assigns values to dialog control variables.
DLGSETCHAR	Function	Assigns values to dialog control variables of type Character.
DLGSETCTRLEVENTHANDLER	Function	Assigns your own event handlers to ActiveX controls in a dialog box.
DLGSETINT	Function	Assigns values to dialog control variables of type Integer.

Name	Routine Type	Description
DLGSETLOG	Function	Assigns values to dialog control variables of type Logical.
DLGSETRETURN	Subroutine	Sets the return value for DLGMODAL.
DLGSETSUB	Function	Assigns procedures (callback routines) to dialog controls.
DLGSETTITLE	Subroutine	Sets the title of a dialog box.
DLGUNINIT	Subroutine	Deallocates memory occupied by an initialized dialog.

COM and Automation Library Routines (W*32, W*64)

The following tables list COM and Automation libary routines.

Programs that use COM routines must access the appropriate libraries with USE IFCOM. Programs that use automation routines must access the appropriate libraries with USE IFAUTO. Some procedures also require the USE IFWINTY module.

The COM and Automation routines are restricted to Windows* systems.

Routine names are shown in mixed case to make the names easier to understand. When writing your applications, you can use any case.

Table 733: Component Object Model (COM) Procedures (USE IFCOM)

Name	Routine Type	Description
COMAddObjectReference	Function	Adds a reference to an object's interface.
COMCLSIDFromProgID[1]	Subroutine	Passes a programmatic identifier and returns the corresponding class identifier.
COMCLSIDFromString[1]	Subroutine	Passes a class identifier string and returns the corresponding class identifier.

Name	Routine Type	Description
COMCreateObjectByGUID[1]	Subroutine	Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.
COMCreateObjectByProgID	Subroutine	Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.
COMGetActiveObjectByGUID[1]	Subroutine	Passes a class identifier and returns a pointer to the interface of a currently active object.
COMGetActiveObjectByProgID	Subroutine	Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.
COMGetFileObject	Subroutine	Passes a file name and returns a pointer to the IDispatch interface of an Automation object that can manipulate the file.
COMInitialize	Subroutine	Initializes the COM library.
COMIsEqualGUID[1]	Function	Determines whether two GUIDs are the same.
COMQueryInterface[1]	Subroutine	Passes an interface identifier and returns a pointer to an object's interface.
COMReleaseObject	Function	Indicates that the program is done with a reference to an object's interface.

Name	Routine Type	Description
COMStringFromGUID[1]	Subroutine	Passes a GUID and returns a string of printable characters.
COMUninitialize	Subroutine	Uninitializes the COM library.

Table 734: Automation Server Procedures (USE IFAUTO)

Name	Routine Type	Description
AUTOAddArg[1]	Subroutine	Passes an argument name and value and adds the argument to the argument list data structure.
AUTOAllocateInvokeArgs	Function	Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.
AUTODeallocateInvokeArgs	Subroutine	Deallocates an argument list data structure.
AUTOGetExceptInfo	Subroutine	Retrieves the exception information when a method has returned an exception status.
AUTOGetProperty[1]	Function	Passes the name or identifier of the property and gets the value of the Automation object's property.
AUTOGetPropertyByID	Function	Passes the member ID of the property and gets the value of the Automation object's property into the argument list's first argument.
AUTOGetPropertyInvokeArgs	Function	Passes an argument list data structure and gets the value of the Automation object's

Name	Routine Type	Description
		property specified in the argument list's first argument.
AUTOInvoke	Function	Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.
AUTOSetProperty[1]	Function	Passes the name or identifier of the property and a value, and sets the value of the Automation object's property.
AUTOSetPropertyByID	Function	Passes the member ID of the property and sets the value of the Automation object's property, using the argument list's first argument.
AUTOSetPropertyInvokeArgs	Function	Passes an argument list data structure and sets the value of the Automation object's property specified in the argument list's first argument.

[1]These routines also require USE IFWINTY.

Miscellaneous Run-Time Library Routines

The following table lists miscellaneous run-time library routines.

Programs that use most of these routines should contain a USE IFCORE statement to obtain the proper interfaces to these routines. You do not need a USE IFCORE statement for for_rtl_init_ and for_rtl_finish_.

Name	Procedure Type	Description
COMMITQQ	Function	Forces the operating system to execute any pending write operations for a file.

Name	Procedure Type	Description
FOR_DESCRIPTOR_ASSIGN[1]	Subroutine	Creates an array descriptor in memory.
FOR_GET_FPE	Function	Returns the current settings of floating-point exception flags.
for_rtl_finish_	Function	Cleans up the Fortran run-time environment.
for_rtl_init_	Function	Initializes the Fortran run-time environment.
FOR_SET_FPE	Function	Sets the floating-point exception flags.
FOR_SET_REENTRANCY	Function	Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits.
GERROR	Subroutine	Returns a message for the last error detected by a Fortran run-time routine.
GETCHARQQ	Function	Returns the next keystroke.
GETEXCEPTIONPTRSQQ[1]	Function	Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.
GETSTRQQ	Function	Reads a character string from the keyboard using buffered input.
PEEKCHARQQ	Function	Checks the buffer to see if a keystroke is waiting.

Name	Procedure Type	Description
PERROR	Subroutine	Sends a message to the standard error stream, preceded by a specified string, for the last detected error.
TRACEBACKQQ	Subroutine	Provides traceback information.

[1] W*32, W*64

Intrinsic Functions Not Allowed as Actual Arguments

This table is now located in Intrinsic Procedures.

A to B

ABORT

Portability Subroutine: *Flushes and closes I/O buffers, and terminates program execution.*

Module

USE IFPORT

Syntax

`CALL ABORT[string]`

string	(Input; optional) Character*(*). Allows you to specify an abort message at program termination. When ABORT is called, "abort:" is written to external unit 0, followed by *string*. If omitted, the default message written to external unit 0 is "abort: Fortran Abort Called."

This subroutine causes the program to terminate and an exit code value of 134 is returned to the program that launched it.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

!The following prints "abort: Fortran Abort Called"

CALL ABORT

!The following prints "abort: Out of here!"

Call ABORT ("Out of here!")
```

See Also

- A to B
- EXIT
- STOP

ABOUTBOXQQ (W*32, W*64)

QuickWin Function: *Specifies the information displayed in the message box that appears when the user selects the About command from a QuickWin application's Help menu.*

Module

USE IFQWIN

Syntax

`result=ABOUTBOXQQ(`*`cstring`*`)`

`cstring` (Input; output) Character*(*). Null-terminated C string.

Results

The value of the result is INTEGER(4). It is zero if successful; otherwise, nonzero.

If your program does not call ABOUTBOXQQ, the QuickWin run-time library supplies a default string.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
        USE IFQWIN
        INTEGER(4) dummy
! Set the About box message
        dummy = ABOUTBOXQQ ('Matrix Multiplier\r      Version 1.0'C)
```

See Also

- A to B

Building Applications: Using QuickWin Overview

Building Applications: Defining an About Box

ABS

Elemental Intrinsic Function (Generic):
Computes an absolute value.

Syntax

`result=ABS(a)`

a (Input) Must be of type integer, real, or complex.

Results

The result has the same type and kind type parameter as *a* except if *a* is complex value, the result type is real. If *a* is an integer or real value, the value of the result is | *a* |; if *a* is a complex value (X, Y), the result is the real value SQRT (X**2 + Y**2).

Specific Name	Argument Type	Result Type
BABS	INTEGER(1)	INTEGER(1)
IIABS[1]	INTEGER(2)	INTEGER(2)
IABS [2]	INTEGER(4)	INTEGER(4)
KIABS	INTEGER(8)	INTEGER(8)
ABS	REAL(4)	REAL(4)

Specific Name	Argument Type	Result Type
DABS	REAL(8)	REAL(8)
QABS	REAL(16)	REAL(16)
CABS [3]	COMPLEX(4)	REAL(4)
CDABS[4]	COMPLEX(8)	REAL(8)
CQABS	COMPLEX(16)	REAL(16)

[1]Or HABS.

[2]Or JIABS. For compatibility with older versions of Fortran, IABS can also be specified as a generic function.

[3]The setting of compiler options specifying real size can affect CABS.

[4]This function can also be specified as ZABS.

Example

ABS (-7.4) has the value 7.4.

ABS ((6.0, 8.0)) has the value 10.0.

The following ABS.F90 program calculates two square roots, retaining the sign:

```
        REAL mag(2), sgn(2), result(2)
        WRITE (*, '(A)') ' Enter two signed magnitudes: '
        READ (*, *) mag
        sgn = SIGN((/1.0, 1.0/), mag) ! transfer the signs to 1.0s
        result = SQRT (ABS (mag))
! Restore the sign by multiplying by -1 or +1:
        result = result * sgn
        WRITE (*, *) result
        END
```

ACCEPT

Statement: *Transfers input data.*

Syntax

Formatted:

```
ACCEPT form[,io-list]
```

Formatted - List-Directed:

```
ACCEPT*[,io-list]
```

Formatted - Namelist:

```
ACCEPTnml
```

form	Is the nonkeyword form of a format specifier (no FMT=).
io-list	Is an I/O list.
*	Is the format specifier indicating list-directed formatting.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

The ACCEPT statement is the same as a formatted, sequential READ statement, except that an ACCEPT statement must never be connected to user-specified I/O units. You can override this restriction by using an environment variable.

Example

In the following example, character data is read from the implicit unit and binary values are assigned to each of the five elements of array CHARAR:

```
     CHARACTER*10 CHARAR(5)

     ACCEPT 200, CHARAR

200  FORMAT (5A10)
```

See Also

- A to B

Building Applications: Logical Devices

ACCESS Function

Portability Function: *Determines if a file exists and how it can be accessed.*

Module

USE IFPORT

Syntax

`result=ACCESS(name,mode)`

name (Input) Character*(*). Name of the file whose accessibility is to be determined.

mode (Input) Character*(*). Modes of accessibility to check for. Must be a character string of length one or greater containing only the characters "r", "w", "x", or "" (a blank). These characters are interpreted as follows.

Character	Meaning
r	Tests for read permission
w	Tests for write permission
x	Tests for execute permission. On Windows* systems, the extension of *name* must be .COM, .EXE, .BAT, .CMD, .PL, .KSH, or .CSH.
(blank)	Tests for existence

The characters within *mode* can appear in any order or combination. For example, wrx and r are legal forms of *mode* and represent the same set of inquiries.

Results

The value of the result is INTEGER(4). It is zero if all inquiries specified by *mode* are true. If either argument is invalid, or if the file cannot be accessed in all of the modes specified, one of the following error codes is returned:

- EACCES: Access denied; the file's permission setting does not allow the specified access
- EINVAL: The mode argument is invalid
- ENOENT: File or path not found

For a list of error codes, see IERRNO.

The *name* argument can contain either forward or backward slashes for path separators.

On Windows* systems, all files are readable. A test for read permission always returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
! checks for read and write permission on the file "DATAFILE.TXT"
J = ACCESS ("DATAFILE.TXT", "rw")
PRINT *, J
! checks whether "DATAFILE.TXT" is executable. It is not, since
! it does not end in .COM, .EXE, .BAT, or .CMD
J = ACCESS ("DATAFILE.TXT","x")
PRINT *, J
```

See Also

- A to B
- INQUIRE
- GETFILEINFOQQ

ACHAR

Elemental Intrinsic Function (Generic):
Returns the character in a specified position of the ASCII character set, even if the processor's default character set is different. It is the inverse of the IACHAR function. In Intel ® Fortran, ACHAR is equivalent to the CHAR function.

Syntax

```
result = ACHAR (i [, kind])
```

i	(Input) Is of type integer.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is character with length 1. If kind is present, the kind parameter of the result is that specified by kind; otherwise, the kind parameter of the result is that of default character. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *i* has a value within the range 0 to 127, the result is the character in position *i* of the ASCII character set; otherwise, it is processor defined. ACHAR (IACHAR(C)) has the value C for any character C capable of representation in the default character set. For a complete list of ASCII character codes, see Character and Key Code Charts.

Example

ACHAR (71) has the value `'G'`.

ACHAR (63) has the value `'?'`.

See Also

- A to B
- CHAR
- IACHAR
- ICHAR

ACOS

Elemental Intrinsic Function (Generic):
Produces the arccosine of x.

Syntax

```
result = ACOS (x)
```

x	(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results

The result type is the same as *x* and is expressed in radians. The value lies in the range 0 to pi.

Specific Name	Argument Type	Result Type
ACOS	REAL(4)	REAL(4)
DACOS	REAL(8)	REAL(8)
QACOS[1]	REAL(16)	REAL(16)

[1]Or QARCOS.

Example

ACOS (0.68032123) has the value .8225955.

ACOSD

Elemental Intrinsic Function (Generic):
Produces the arccosine of x.

Syntax

```
result = ACOSD (x)
```

x	(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results

The result type is the same as *x* and is expressed in degrees. The value lies in the range –90 to 90 degrees.

Specific Name	Argument Type	Result Type
ACOSD	REAL(4)	REAL(4)
DACOSD	REAL(8)	REAL(8)
QACOSD	REAL(16)	REAL(16)

Example

ACOSD (0.886579) has the value 27.55354.

ACOSH

Elemental Intrinsic Function (Generic):
Produces the hyperbolic arccosine of x.

Syntax

```
result = ACOSH (x)
```

x	(Input) Must be of type real and must be greater than or equal to 1.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
ACOSH	REAL(4)	REAL(4)
DACOSH	REAL(8)	REAL(8)
QACOSH	REAL(16)	REAL(16)

Example

ACOSH (1.0) has the value 0.0.

ACOSH (180.0) has the value 5.8861.

ADJUSTL

Elemental Intrinsic Function (Generic): *Adjusts a character string to the left, removing leading blanks and inserting trailing blanks.*

Syntax

```
result = ADJUSTL (string)
```

string	(Input) Must be of type character.

Results

The result type is character with the same length and kind parameter as *string*. The value of the result is the same as *string*, except that any leading blanks have been removed and inserted as trailing blanks.

Example

```
CHARACTER(16) STRING  STRING= ADJUSTL('   Fortran 90   ') ! returns 'Fortran 90   '
ADJUSTL ('    SUMMERTIME')     ! has the value 'SUMMERTIME    '
```

See Also

- A to B
- ADJUSTR

ADJUSTR

Elemental Intrinsic Function (Generic): *Adjusts a character string to the right, removing trailing blanks and inserting leading blanks.*

Syntax

```
result = ADJUSTR (string)
```

string (Input) Must be of type character.

Results

The result type is character with the same length and kind parameter as *string*.

The value of the result is the same as *string*, except that any trailing blanks have been removed and inserted as leading blanks.

Example

```
CHARACTER(16) STRING
STRING= ADJUSTR('   Fortran 90   ') ! returns '     Fortran 90'
ADJUSTR ('SUMMERTIME    ')     ! has the value '    SUMMERTIME'
```

See Also

- A to B
- ADJUSTL

AIMAG

Elemental Intrinsic Function (Generic):
Returns the imaginary part of a complex number.
This function can also be specified as IMAG.

Syntax

```
result = AIMAG (z)
```

z (Input) Must be of type complex.

Results

The result type is real with the same kind parameter as *z*. If *z* has the value (x, y), the result has the value y.

Specific Name	Argument Type	Result Type
AIMAG [1]	COMPLEX(4)	REAL(4)
DIMAG	COMPLEX(8)	REAL(8)
QIMAG	COMPLEX(16)	REAL(16)

[1] The setting of compiler options specifying real size can affect AIMAG.

To return the real part of complex numbers, use REAL.

Example

AIMAG ((4.0, 5.0)) has the value 5.0.

The program AIMAG.F90 applies the quadratic formula to a polynomial and allows for complex results:

```
REAL a, b, c
COMPLEX ans1, ans2, d
WRITE ( *, 100)
100 FORMAT ('  Enter A, b, and c of the  ',   &
                          'polynomial ax**2 + bx + c:  '\)
READ (*, *) a, b, c
d = CSQRT (CMPLX (b**2 - 4.0*a*c))  ! d is either:
                                    !  0.0 + i root, or
                                    !  root + i 0.0
ans1 = (-b + d) / (2.0 * a)
ans2 = (-b + d) / (2.0 * a)
WRITE (*, 200)
200 FORMAT (/ '  The roots are:'  /)
WRITE (*, 300) REAL(ans1), AIMAG(ans1), &
                  REAL(ans2), AIMAG(ans2)
300 FORMAT (' X = ', F10.5, ' + i', F10.5)
END
```

See Also

- A to B
- CONJG
- DBLE

AINT

Elemental Intrinsic Function (Generic):
Truncates a value to a whole number.

Syntax

```
result = AINT (a [,kind])
```

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is real. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter is that of *a*.

The result is defined as the largest integer whose magnitude does not exceed the magnitude of *a* and whose sign is the same as that of *a*. If | *a* | is less than 1, AINT(*a*) has the value zero.

Specific Name	Argument Type	Result Type
AINT	REAL(4)	REAL(4)
DINT	REAL(8)	REAL(8)
QINT	REAL(16)	REAL(16)

To round rather than truncate, use ANINT.

Example

AINT (3.678) has the value 3.0.

AINT (-1.375) has the value -1.0.

```
REAL r1, r2
REAL(8) r3(2)
r1 = AINT(2.6)   ! returns the value 2.0
r2 = AINT(-2.6)  ! returns the value -2.0
r3 = AINT((/1.3, 1.9/), KIND = 8)       ! returns the values
                                        ! (1.0D0, 1.0D0)
```

See Also

- A to B
- ANINT

ALARM

Portability Function: *Causes a subroutine to begin execution after a specified amount of time has elapsed.*

Module

USE IFPORT

Syntax

`result = ALARM (time,proc)`

time	(Input) Integer. Specifies the time delay, in seconds, between the call to ALARM and the time when *proc* is to begin execution. If *time* is 0, the alarm is turned off and no routine is called.
proc	(Input) Name of the procedure to call. The procedure takes no arguments and must be declared EXTERNAL.

Results

The return value is INTEGER(4). It is zero if no alarm is pending. If an alarm is pending (has already been set by a previous call to ALARM), it returns the number of seconds remaining until the previously set alarm is to go off, rounded up to the nearest second.

After ALARM is called and the timer starts, the calling program continues for *time* seconds. The calling program then suspends and calls *proc*, which runs in another thread. When *proc* finishes, the alarm thread terminates, the original thread resumes, and the calling program resets the alarm. Once the alarm goes off, it is disabled until set again.

If *proc* performs I/O or otherwise uses the Fortran library, you need to compile it with one of the multithread libraries.

The thread that *proc* runs in has a higher priority than any other thread in the process. All other threads are essentially suspended until *proc* terminates, or is blocked on some other event, such as I/O.

No alarms can occur after the main process ends. If the main program finishes or any thread executes an EXIT call, than any pending alarm is deactivated before it has a chance to run.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) numsec, istat
EXTERNAL subprog
numsec = 4
write *, "subprog will begin in ", numsec, " seconds"
ISTAT = ALARM (numsec, subprog)
```

See Also

- A to B
- RUNQQ
- Creating Multithread Applications Overview

ALIAS Directive

General Compiler Directive: *Declares alternate external names for external subprograms.*

Syntax

*c*DEC$ ALIAS *internal-name*,*external-name*

c	Is a c, C, !, or *. (See Syntax Rules for Compiler Directives.)
internal-name	The name of the subprogram as used in the current program unit.
external-name	A name or a character constant, delimited by apostrophes or quotation marks.

Description

If a name is specified, the name (in uppercase) is used as the external name for the specified *internal-name*. If a character constant is specified, it is used as is; the string is not changed to uppercase, nor are blanks removed.

The ALIAS directive affects only the external name used for references to the specified *internal-name*.

Names that are not acceptable to the linker will cause link-time errors.

See Also

- A to B

- ATTRIBUTES- ALIAS option
- General Compiler Directives

ALL

Transformational Intrinsic Function (Generic):
Determines if `all` values are true in an entire array or in a specified dimension of an array.

Syntax

`result = ALL (mask [,dim])`

mask	(Input) Must be a logical array.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *mask*.

Results

The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true only if all elements of *mask* are true, or *mask* has size zero. The result has the value false if any element of *mask* is false.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result is true only if all elements in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) are true.

Example

```
        LOGICAL mask( 2, 3), AR1(3), AR2(2)
        mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., .FALSE., &
                        .FALSE./),(/2,3/))
! mask is true false false
! true true false
        AR1 = ALL(mask,DIM = 1) ! evaluates the elements column by
                ! column yielding [true false false]
        AR2 = ALL(mask,DIM = 2) ! evaluates the elements row by row
                ! yielding [false false].
```

ALL ((/.TRUE., .FALSE., .TRUE./)) has the value false because some elements of MASK are not true.

ALL ((/.TRUE., .TRUE., .TRUE./)) has the value true because *all* elements of MASK are true.

A is the array

```
[ 1  5  7 ]
[ 3  6  8 ]
```

and B is the array

```
[ 0  5  7 ]
[ 2  6  9 ].
```

ALL (A .EQ. B, DIM=1) tests to see if all elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, false) because only the second column has elements that are all equal.

ALL (A .EQ. B, DIM=2) tests to see if all elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (false, false) because each row has some elements that are not equal.

See Also

- A to B
- ANY
- COUNT

ALLOCATABLE

Statement and Attribute: *Specifies that an array is an allocatable array with a deferred shape. The shape of an allocatable array is determined when an ALLOCATE statement is executed, dynamically allocating space for the array.*

Syntax

The ALLOCATABLE attribute can be specified in a type declaration statement or an ALLOCATABLE statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] ALLOCATABLE [att-ls,] :: a[(d-spec)] [, a[(d-spec)]] ...
```

Statement:

```
ALLOCATABLE [::] a[(d-spec)] [, a[(d-spec)]] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
a	Is the name of the allocatable array.
d-spec	Is a deferred-shape specification (: [, :] ...). Each colon represents a dimension of the array.

Description

If the array is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

When the allocatable array is no longer needed, it can be deallocated by execution of a DEALLOCATE statement.

An allocatable array cannot be specified in a COMMON, EQUIVALENCE, DATA, or NAMELIST statement.

Allocatable arrays are not saved by default. If you want to retain the values of an allocatable array across procedure calls, you must specify the SAVE attribute for the array.

Example

```
!Method for creating and allocating deferred-shape arrays.
INTEGER, ALLOCATABLE :: matrix(:,:)
REAL, ALLOCATABLE    :: vector(:)
...
ALLOCATE(matrix(3,5),vector(-2:N+2))
...
```

The following example shows a type declaration statement specifying the ALLOCATABLE attribute:

```
REAL, ALLOCATABLE :: Z(:, :, :)
```

The following is an example of the ALLOCATABLE statement:

```
REAL A, B(:)
ALLOCATABLE :: A(:,:), B
```

See Also

- A to B
- Type declaration statements
- Compatible attributes
- DEALLOCATE
- Arrays
- Allocation of Allocatable Arrays
- SAVE

ALLOCATE

Statement: *Dynamically creates storage for allocatable arrays and pointer targets. The storage space allocated is uninitialized.*

Syntax

ALLOCATE (*object*[(*s-spec*)] [,*object*[(*s-spec*[,*s-spec*]...)]]...[, *alloc-opt*])

object	Is the object to be allocated. It is a variable name or structure component, and must be a pointer or allocatable array. The object can be of type character with zero length.

s-spec	Is a shape specification in the form [lower-bound:]upper-bound. Each bound must be a scalar integer expression. The number of shape specifications must be the same as the rank of the *object*.	
alloc-opt	(Output) Is one of the following:	
	STAT=*sv*	*sv* is a scalar integer variable in which the status of the allocation is stored.
	ERRMSG=*ev*	*ev* is a scalar default character value in which an error condition is stored if such a condition occurs.

Description

A bound in *s-spec* must not be an expression containing an array inquiry function whose argument is any allocatable object in the same ALLOCATE statement; for example, the following is not permitted:

```
INTEGER ERR
INTEGER, ALLOCATABLE :: A(:), B(:)
...
ALLOCATE(A(10:25), B(SIZE(A)), STAT=ERR)  ! A is invalid as an argument
                                          !   to function SIZE
```

If a STAT variable or ERRMSG variable is specified, it must not be allocated in the ALLOCATE statement in which it appears. If the allocation is successful, the variable is set to zero. If the allocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error); the ERRMSG variable contains the error condition. If no STAT variable is specified and an error condition occurs, program execution terminates.

To release the storage for an allocated array, use DEALLOCATE.

To determine whether an allocatable array is currently allocated, use the ALLOCATED intrinsic function.

To determine whether a pointer is currently associated with a target, use the ASSOCIATED intrinsic function.

Example

```
!Method for creating and allocating deferred shape arrays.
INTEGER,ALLOCATABLE::matrix(:,:)
REAL, ALLOCATABLE:: vector(:)
. . .
ALLOCATE (matrix(3,5),vector(-2:N+2))
. . .
```

The following shows another example of the ALLOCATE statement:

```
INTEGER J, N, ALLOC_ERR
REAL, ALLOCATABLE :: A(:), B(:,:)
...
ALLOCATE(A(0:80), B(-3:J+1, N), STAT = ALLOC_ERR)
```

See Also

- A to B
- ALLOCATABLE
- ALLOCATED
- DEALLOCATE
- ASSOCIATED
- POINTER
- Dynamic Allocation
- Pointer Assignments

ALLOCATED

Inquiry Intrinsic Function (Generic): *Indicates whether an allocatable array is currently allocated.*

Syntax

`result = ALLOCATED (`*`array`*`)`

`array`	(Input) Must be an allocatable array.

Results

The result is a scalar of type default logical.

The result has the value true if *array* is currently allocated, false if *array* is not currently allocated, or undefined if its allocation status is undefined.

Example

```
REAL, ALLOCATABLE :: A(:)

...

IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

Consider the following:

```
REAL, ALLOCATABLE, DIMENSION (:,:,:) :: E

PRINT *, ALLOCATED (E)       ! Returns the value false

ALLOCATE (E (12, 15, 20))

PRINT *, ALLOCATED (E)       ! Returns the value true
```

See Also

- A to B
- ALLOCATABLE
- ALLOCATE
- DEALLOCATE
- Arrays
- Dynamic Allocation

AND

Elemental Intrinsic Function (Generic)

Example

```
INTEGER(1) i, m
INTEGER result
INTEGER(2) result2
i = 1
m = 3
result = AND(i,m) ! returns an integer of default type
        ! (INTEGER(4) unless reset by user) whose
        ! value = 1
result2 = AND(i,m) ! returns an INTEGER(2) with value = 1
```

See Also

- A to B
- IAND

ANINT

Elemental Intrinsic Function (Generic):
Calculates the nearest whole number.

Syntax

`result = ANINT (a[,kind] )`

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of *a*. If *a* is greater than zero, ANINT (*a*) has the value AINT (*a* + 0.5); if *a* is less than or equal to zero, ANINT (*a*) has the value AINT (*a* - 0.5).

Specific Name	Argument Type	Result Type
ANINT	REAL(4)	REAL(4)
DNINT	REAL(8)	REAL(8)
QNINT	REAL(16)	REAL(16)

To truncate rather than round, use AINT.

Example

ANINT (3.456) has the value 3.0.

ANINT (-2.798) has the value -3.0.

Consider the following:

```
       REAL r1, r2
       r1 = ANINT(2.6)      ! returns the value 3.0
       r2 = ANINT(-2.6)     ! returns the value -3.0
! ANINT.F90 Calculates and adds tax to a purchase amount.
       REAL amount, taxrate, tax, total
       taxrate = 0.081
       amount = 12.99
       tax = ANINT (amount * taxrate * 100.0) / 100.0
       total = amount + tax
       WRITE (*, 100) amount, tax, total
  100  FORMAT ( 1X, 'AMOUNT', F7.2 /
      + 1X, 'TAX ', F7.2 /
      + 1X, 'TOTAL ', F7.2)
       END
```

See Also

- A to B
- NINT

ANY

Transformational Intrinsic Function (Generic):
Determines if any value is true in an entire array or in a specified dimension of an array.

Syntax

```
result = ANY (mask [, dim])
```

mask	(Input) Must be a logical array.
dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.

Results

The result is an array or a scalar of type logical.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result is true if any elements of *mask* are true. The result has the value false if no element of *mask* is true, or *mask* has size zero.

An array result has the same type and kind parameters as *mask*, and a rank that is one less than *mask*. Its shape is (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *mask*.

Each element in an array result is true if any elements in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) are true.

Example

```
    LOGICAL mask( 2, 3), AR1(3), AR2(2)

    logical, parameter :: T = .true.
    logical, parameter :: F = .false.
    DATA mask /T, T, F, T, F, F/

! mask is     true false false

!            true true false

    AR1 = ANY(mask,DIM = 1)  ! evaluates the elements column by

                             !   column yielding [true true false]

    AR2 = ANY(mask,DIM = 2)  ! evaluates the elements row by row

                             !   yielding [true true]
```

ANY ((/.FALSE., .FALSE., .TRUE./)) has the value true because one element is true.

A is the array

```
[ 1  5  7 ]
[ 3  6  8 ]
```

and B is the array

```
[ 0  5  7 ]
[ 2  6  9 ].
```

ANY (A .EQ. B, DIM=1) tests to see if any elements in each column of A are equal to the elements in the corresponding column of B. The result has the value (false, true, true) because the second and third columns have at least one element that is equal.

ANY (A .EQ. B, DIM=2) tests to see if any elements in each row of A are equal to the elements in the corresponding row of B. The result has the value (true, true) because each row has at least one element that is equal.

See Also

- A to B
- ALL
- COUNT

APPENDMENUQQ (W*32, W*64)

QuickWin Function: *Appends a menu item to the end of a menu and registers its callback subroutine.*

Module

USE IFQWIN

Syntax

`result = APPENDMENUQQ (menuID, flags, text,routine)`

menuID	(Input) INTEGER(4). Identifies the menu to which the item is appended, starting with 1 as the leftmost menu.
flags	(Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see the Results section below). The following constants are available: • $MENUGRAYED - Disables and grays out the menu item.

- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

text (Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, 'WORDS OF TEXT'C.

routine (Input) EXTERNAL. Callback subroutine that is called if the menu item is selected. All routines take a single LOGICAL parameter that indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.
- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.
- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from the current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.

- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.
- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is logical. It is .TRUE. if successful; otherwise, .FALSE..

You do not need to specify a menu item number, because APPENDMENUQQ always adds the new item to the bottom of the menu list. If there is no item yet for a menu, your appended item is treated as the top-level menu item (shown on the menu bar), and *text* becomes the menu title. APPENDMENUQQ ignores the callback routine for a top-level menu item if there are any other menu items in the menu. In this case, you can set *routine* to NUL.

If you want to insert a menu item into a menu rather than append to the bottom of the menu list, use INSERTMENUQQ.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to APPENDMENUQQ as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY (ALT+R in the example) as an alternative to selecting the item with the mouse.

For more information about customizing QuickWin menus, see Building Applications: Using QuickWin Overview.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
      USE IFQWIN
      LOGICAL(4) result
      CHARACTER(25) str
      ...
! Append two items to the bottom of the first (FILE) menu
      str    = '&Add to File Menu'C ! 'A' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
      str    = 'Menu Item &2b'C ! '2' is a quick-access key
      result = APPENDMENUQQ(1, $MENUENABLED, str, WINCASCADE)
! Append an item to the bottom of the second (EDIT) menu
      str    = 'Add to Second &Menu'C ! 'M' is a quick-access key
      result = APPENDMENUQQ(2, $MENUENABLED, str, WINTILE)
```

See Also

- A to B
- INSERTMENUQQ
- DELETEMENUQQ
- MODIFYMENUFLAGSQQ
- MODIFYMENUROUTINEQQ
- MODIFYMENUSTRINGQQ

ARC, ARC_W (W*32, W*64)

Graphics Functions: *Draw elliptical arcs using the current graphics color.*

Module

USE IFQWIN

Syntax

```
result = ARC (x1,y1,x2,y2,x3,y3,x4,y4)
result = ARC_W (wx1,wy1,wx2,wy2,wx3,wy3,wx4,wy4)
```

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
x3, y3	(Input) INTEGER(2). Viewport coordinates of start vector.
x4, y4	(Input) INTEGER(2). Viewport coordinates of end vector.
wx1,wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.
wx3, wy3	(Input) REAL(8). Window coordinates of start vector.
wx4, wy4	(Input) REAL(8). Window coordinates of end vector.

Results

The result type is INTEGER(2). It is nonzero if successful; otherwise, 0. If the arc is clipped or partially out of bounds, the arc is considered successfully drawn and the return is 1. If the arc is drawn completely out of bounds, the return is 0.

The center of the arc is the center of the bounding rectangle defined by the points (*x1*, *y1*) and (*x2*, *y2*) for ARC and (*wx1*, *wy1*) and (*wx2*, *wy2*) for ARC_W.

The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for ARC and (*wx3*, *wy3*) for ARC_W. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for ARC and (*wx4*, *wy4*) for ARC_W.

ARC uses the view-coordinate system. ARC_W uses the window-coordinate system. In each case, the arc is drawn using the current color.

NOTE. The ARC routine described here is a QuickWin graphics routine. If you are trying to use the Microsoft* Platform SDK version of the Arc routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Arc.

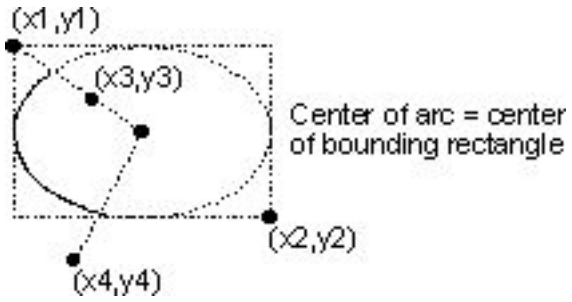
Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

This program draws the arc shown below.

```
USE IFQWIN

INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4

x1 = 80; y1 = 50

x2 = 240; y2 = 150

x3 = 120; y3 = 75

x4 = 90; y4 = 180

status = ARC( x1, y1, x2, y2, x3, y3, x4, y4 )

END
```



ASIN

Elemental Intrinsic Function (Generic):
Produces the arcsine of x.

Syntax

`result = ASIN (x)`

x	(Input) Must be of type real. The | x | must be less than or equal to 1.

Results

The result type is the same as x and is expressed in radians. The value lies in the range -pi/2 to pi/2.

Specific Name	Argument Type	Result Type
ASIN	REAL(4)	REAL(4)
DASIN	REAL(8)	REAL(8)
QASIN	REAL(16)	REAL(16)

Example

ASIN (0.79345021) has the value 0.9164571.

ASIND

Elemental Intrinsic Function (Generic):
Produces the arcsine of x.

Syntax

```
result = ASIND (x)
```

x	(Input) Must be of type real. The | *x* | must be less than or equal to 1.

Results

The result type is the same as *x* and is expressed in degrees. The value lies in the range –90 to 90 degrees.

Specific Name	Argument Type	Result Type
ASIND	REAL(4)	REAL(4)
DASIND	REAL(8)	REAL(8)
QASIND	REAL(16)	REAL(16)

Example

ASIND (0.2467590) has the value 14.28581.

ASINH

Elemental Intrinsic Function (Generic):
Produces the hyperbolic arcsine of x.

Syntax

```
result = ASINH (x)
```

x	(Input) Must be of type real.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
ASINH	REAL(4)	REAL(4)
DASINH	REAL(8)	REAL(8)
QASINH	REAL(16)	REAL(16)

Example

ASINH (1.0) has the value -0.88137.

ASINH (180.0) has the value 5.88611.

ASSIGN - Label Assignment

Statement: *Assigns a statement label value to an integer variable. This feature has been deleted in Fortran 95; it was obsolescent in Fortran 90. Intel® Fortran fully supports features deleted in Fortran 95.*

Syntax

```
ASSIGN label TO var
```

label	Is the label of a branch target or FORMAT statement in the same scoping unit as the ASSIGN statement.
var	Is a scalar integer variable.

When an ASSIGN statement is executed, the statement label is assigned to the integer variable. The variable is then undefined as an integer variable and can only be used as a label (unless it is later redefined with an integer value).

The ASSIGN statement must be executed before the statements in which the assigned variable is used.

Indirect branching through integer variables makes program flow difficult to read, especially if the integer variable is also used in arithmetic operations. Using these statements permits inconsistent usage of the integer variable, and can be an obscure source of error. The ASSIGN statement was used to simulate internal procedures, which now can be coded directly.

Example

The value of a label is not the same as its number; instead, the label is identified by a number assigned by the compiler. In the following example, 400 is the label number (not the value) of IVBL:

```
ASSIGN 400 TO IVBL
```

Variables used in ASSIGN statements are not defined as integers. If you want to use a variable defined by an ASSIGN statement in an arithmetic expression, you must first define the variable by a computational assignment statement or by a READ statement, as in the following example:

```
IVBL = 400
```

The following example shows ASSIGN statements:

```
INTEGER ERROR

...

ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR
```

Note that NSTART and KSTOP are integer variables implicitly, but ERROR must be previously declared as an integer variable.

The following statement associates the variable NUMBER with the statement label 100:

```
ASSIGN 100 TO NUMBER
```

If an arithmetic operation is subsequently performed on variable NUMBER (such as follows), the run-time behavior is unpredictable:

```
NUMBER = NUMBER + 1
```

To return NUMBER to the status of an integer variable, you can use the following statement:

```
NUMBER = 10
```

This statement dissociates NUMBER from statement 100 and assigns it an integer value of 10. Once NUMBER is returned to its integer variable status, it can no longer be used in an assigned GO TO statement.

See Also

- A to B
- Assignment: intrinsic
- Obsolescent Features in Fortran 90

Assignment(=) - Defined Assignment

Statement: *An interface block that defines generic assignment. The only procedures allowed in the interface block are subroutines that can be referenced as defined assignments.*

Syntax

The initial line for such an interface block takes the following form:

```
INTERFACE ASSIGNMENT (=)
```

Description

The subroutines within the interface block must have two nonoptional arguments, the first with intent OUT or INOUT, and the second with intent IN.

A defined assignment is treated as a reference to a subroutine. The left side of the assignment corresponds to the first dummy argument of the subroutine; the right side of the assignment corresponds to the second argument.

The ASSIGNMENT keyword extends or redefines an assignment operation if both sides of the equal sign are of the same derived type.

Defined elemental assignment is indicated by specifying ELEMENTAL in the SUBROUTINE statement.

Any procedure reference involving generic assignment must be resolvable to one specific procedure; it must be unambiguous. For more information, see Unambiguous Generic Procedure References.

Example

The following is an example of a procedure interface block defining assignment:

```
INTERFACE ASSIGNMENT (=)
  SUBROUTINE BIT_TO_NUMERIC (NUM, BIT)
    INTEGER, INTENT(OUT) :: NUM
    LOGICAL, INTENT(IN)  :: BIT(:)
  END SUBROUTINE BIT_TO_NUMERIC
  SUBROUTINE CHAR_TO_STRING (STR, CHAR)
    USE STRING_MODULE                   ! Contains definition of type STRING
    TYPE(STRING), INTENT(OUT) :: STR    ! A variable-length string
    CHARACTER(*), INTENT(IN)  :: CHAR
  END SUBROUTINE  CHAR_TO_STRING
END  INTERFACE
```

The following example shows two equivalent ways to reference subroutine BIT_TO_NUMERIC:

```
  CALL BIT_TO_NUMERIC(X, (NUM(I:J)))
  X = NUM(I:J)
```

The following example shows two equivalent ways to reference subroutine CHAR_TO_STRING:

```
  CALL CHAR_TO_STRING(CH, '432C')
  CH = '432C'
!Converting circle data to interval data.
module mod1
TYPE CIRCLE
        REAL radius, center_point(2)
END TYPE CIRCLE
TYPE INTERVAL
        REAL lower_bound, upper_bound
END TYPE INTERVAL
CONTAINS
        SUBROUTINE circle_to_interval(I,C)
          type (interval),INTENT(OUT)::I
          type (circle),INTENT(IN)::C
!Project circle center onto the x=-axis
!Note: the length of the interval is the diameter of the circle
                I%lower_bound = C%center_point(1) - C%radius
                I%upper_bound = C%center_point(1) + C%radius
        END SUBROUTINE circle_to_interval
end module mod1
PROGRAM assign
use mod1
TYPE(CIRCLE) circle1
TYPE(INTERVAL) interval1
INTERFACE ASSIGNMENT(=)
        module procedure circle_to_interval
END INTERFACE
!Begin executable part of program
```

```
  circle1%radius = 2.5
  circle1%center_point = (/3.0,5.0/)
  interval1 = circle1
. . .
END PROGRAM
```

See Also

- A to B
- INTERFACE
- Assignment Statements

Assignment - Intrinsic Computational

Statement: *Assigns a value to a nonpointer variable. In the case of pointers, intrinsic assignment is used to assign a value to the target associated with the pointer variable. The value assigned to the variable (or target) is determined by evaluation of the expression to the right of the equal sign.*

Syntax

variable=expression

variable	Is the name of a scalar or array of intrinsic or derived type (with no defined assignment). The array cannot be an assumed-size array, and neither the scalar nor the array can be declared with the PARAMETER or INTENT(IN) attribute.
expression	Is of intrinsic type or the same derived type as *variable*. Its shape must conform with *variable*. If necessary, it is converted to the same type and kind as *variable*.

Description

Before a value is assigned to the variable, the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement.

NOTE. When the run-time system assigns a value to a scalar integer or character variable and the variable is shorter than the value being assigned, the assigned value may be truncated and significant bits (or characters) lost. This truncation can occur without warning, and can cause the run-time system to pass incorrect information back to the program.

If the variable is a pointer, it must be associated with a definable target. The shape of the target and expression must conform and their type and kind parameters must match.

If the cDEC$ NOSTRICT compiler directive (the default) is in effect, then you can assign a character expression to a noncharacter variable, and a noncharacter variable or array element (but not an expression) to a character variable.

Example

```
    REAL a, b, c
    LOGICAL abigger
    CHARACTER(16) assertion
    c = .01
    a = SQRT (c)
    b = c**2
    assertion = 'a > b'
    abigger = (a .GT. b)
    WRITE (*, 100) a, b
100 FORMAT (' a =', F7.4, ' b =', F7.4)
    IF (abigger) THEN
            WRITE (*, *) assertion, ' is true.'
    ELSE
            WRITE (*, *) assertion, ' is false.'
    END IF
    END
! The program above has the following output:
!     a =    .1000     b =   .0001         a > b is true.
! The following code shows legal and illegal
! assignment statements:
!    INTEGER i, j
    REAL rone(4), rtwo(4), x, y
    COMPLEX z
    CHARACTER name6(6), name8(8)
    i         = 4
    x         = 2.0
    z         = (3.0, 4.0)
```

```
    rone(1) = 4.0
    rone(2) = 3.0
    rone(3) = 2.0
    rone(4) = 1.0
    name8   = 'Hello,'
! The following assignment statements are legal:
    i     = rone(2); j = rone(i); j = x
    y     = x; y = z; y = rone(3); rtwo = rone; rtwo = 4.7
    name6 = name8
! The following assignment statements are illegal:
    name6 = x + 1.0; int = name8//'test'; y = rone
    END
```

See Also

- A to B
- Assignment: defined
- NOSTRICT directive

ASSOCIATED

Inquiry Intrinsic Function (Generic): *Returns the association status of its pointer argument or indicates whether the pointer is associated with the target.*

Syntax

`result = ASSOCIATED (`*`pointer`* `[, `*`target`*`])`

pointer	(Input) Must be a pointer. It can be of any data type. The pointer association status must be defined.
target	(Input; optional) Must be a pointer or target. If it is a pointer, the pointer association status must be defined.

Results

The result is a scalar of type default logical. The setting of compiler options specifying integer size can affect this function.

If only *pointer* appears, the result is true if it is currently associated with a target; otherwise, the result is false.

If *target* also appears and is a target, the result is true if *pointer* is currently associated with *target*; otherwise, the result is false.

If *target* is a pointer, the result is true if both *pointer* and *target* are currently associated with the same target; otherwise, the result is false. (If either *pointer* or *target* is disassociated, the result is false.)

Example

```
REAL C (:), D(:), E(5)
POINTER C, D
TARGET E
LOGICAL STATUS
C => E                        ! pointer assignment
D => E                        ! pointer assignment
STATUS = ASSOCIATED(C)        ! returns TRUE; C is associated
STATUS = ASSOCIATED(C, E)     ! returns TRUE; C is associated with E
STATUS = ASSOCIATED (C, D)    ! returns TRUE; C and D are associated
                              ! with the same target
```

Consider the following:

```
REAL, TARGET, DIMENSION (0:50) :: TAR
REAL, POINTER, DIMENSION (:) :: PTR
PTR => TAR
PRINT *, ASSOCIATED (PTR, TAR)      ! Returns the value true
```

The subscript range for PTR is 0:50. Consider the following pointer assignment statements:

```
(1) PTR => TAR (:)

(2) PTR => TAR (0:50)

(3) PTR => TAR (0:49)
```

For statements 1 and 2, ASSOCIATED (PTR, TAR) is true because TAR has not changed (the subscript range for PTR in both cases is 1:51, following the rules for deferred-shape arrays). For statement 3, ASSOCIATED (PTR, TAR) is false because the upper bound of TAR has changed.

Consider the following:

```
REAL, POINTER, DIMENSION (:) :: PTR2, PTR3

ALLOCATE (PTR2 (0:15))

PTR3 => PTR2

PRINT *, ASSOCIATED (PTR2, PTR3)       ! Returns the value true

...

NULLIFY (PTR2)

NULLIFY (PTR3)

PRINT *, ASSOCIATED (PTR2, PTR3)       ! Returns the value false
```

See Also

- A to B
- ALLOCATED
- POINTER
- TARGET
- Pointer Assignments

ASSUME_ALIGNED

General Compiler Directive: *Specifies that an entity in memory is aligned.*

Syntax

*c*DEC$ ASSUME_ALIGNED *address1*:*n1* [, *address2*:*n2*]...

c Is a c, C, !, or *. (See Syntax Rules for Compiler Directives.)

address	A memory reference. It can be of any data type, kind, or rank. It cannot be any of the following: • An entity in COMMON (or an entity EQUIVALENCEd to something in COMMON) • A component of a variable of derived type or a record field reference • An entity accessed by use or host association
n	A positive integer initialization expression. Its value must be a power of 2 between 1 and 256, that is, 1, 2, 4, 8, 16,3 2, 64, 128, 256.

If you specify more than one *address*:*n* item, they must be separated by a comma.

If *address* is a Cray POINTER or it has the POINTER attribute, it is the POINTER and not the pointee or the TARGET that is assumed aligned.

If you specify an invalid value for *n*, an error message is displayed.

See Also

- A to B
- PREFETCH
- ATTRIBUTES ALIGN

ASYNCHRONOUS

Statement and Attribute: *Specifies that a variable can be used for asynchronous input and output.*

Syntax

The ASYNCHRONOUS attribute can be specified in a type declaration statement or an ASYNCHRONOUS statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] ASYNCHRONOUS [att-ls,] :: var [, var] ...
```

Statement:

```
ASYNCHRONOUS [::] var [, var] ...
```

type	Is a data type specifier.

att-ls	Is an optional list of attribute specifiers.
var	Is the name of a variable.

Description

Asynchronous I/O, or non-blocking I/O, allows a program to continue processing data while the I/O operation is performed in the background.

A variable can have the ASYNCHRONOUS attribute in a particular scoping unit without necessarily having it in other scoping units. If an object has the ASYNCHRONOUS attribute, then all of its subobjects also have the ASYNCHRONOUS attribute.

The ASYNCHRONOUS attribute can also be implied by use of a variable in an asynchronous READ or WRITE statement.

Examples

The following example shows how the ASYNCHRONOUS attribute can be applied in an OPEN and READ statement.

```
program test
integer, asynchronous, dimension(100) :: array
open (unit=1,file='asynch.dat',asynchronous='YES',&
  form='unformatted')
write (1) (i,i=1,100)
rewind (1)
read (1,asynchronous='YES') array
wait(1)
write (*,*) array(1:10)
end
```

See Also

- A to B
- Type Declarations
- Compatible attributes

ATAN

Elemental Intrinsic Function (Generic):
Produces the arctangent of x.

Syntax

```
result = ATAN (x)
```

x	(Input) Must be of type real.

Results

The result type is the same as *x* and is expressed in radians. The value lies in the range -pi/2 to pi/2.

Specific Name	Argument Type	Result Type
ATAN	REAL(4)	REAL(4)
DATAN	REAL(8)	REAL(8)
QATAN	REAL(16)	REAL(16)

Example

ATAN (1.5874993) has the value 1.008666.

ATAN2

Elemental Intrinsic Function (Generic):
Produces an arctangent (inverse tangent). The result is the principal value of the argument of the nonzero complex number (x, y).

Syntax

```
result = ATAN2 (y,x)
```

y	(Input) Must be of type real.
x	(Input) Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results

The result type is the same as x and is expressed in radians. The value lies in the range -pi <= ATAN2 (y, x) <= pi.

If x is not zero, the result is approximately equal to the value of arctan (y/ x).

If y > zero, the result is positive.

If y < zero, the result is negative.

If y is zero and x > zero, the result is y (so for x>0, ATAN2 ((+0.0), x) is +0.0 and ATAN2 ((-0.0), x) is -0.0).

If y is a positive real zero and x < zero, the result is pi.

If y is a negative real zero and x < zero, the result is -pi.

If x is a positive real zero, the result is pi/2.

If y is a negative real zero, the result is -pi/2.

Specific Name	Argument Type	Result Type
ATAN2	REAL(4)	REAL(4)
DATAN2	REAL(8)	REAL(8)
QATAN2	REAL(16)	REAL(16)

Example

ATAN2 (2.679676, 1.0) has the value 1.213623.

If Y is an array that has the value

```
[  1   1 ]
[ -1  -1 ]
```

and X is an array that has the value

```
[ -1  1 ]
[ -1  1 ],
```

then ATAN2 (Y, X) is

$$\begin{bmatrix} \frac{3pi}{4} & \frac{pi}{4} \\ \frac{-3pi}{4} & \frac{-pi}{4} \end{bmatrix}$$

ATAN2D

Elemental Intrinsic Function (Generic):
Produces an arctangent. The result is the principal value of the argument of the nonzero complex number (x, y).

Syntax

```
result = ATAN2D (y,x)
```

y	(Input) Must be of type real.
x	(Input) Must have the same type and kind parameters as *y*. If *y* has the value zero, *x* cannot have the value zero.

Results

The result type is the same as *x* and is expressed in degrees. The value lies in the range -180 degrees to 180 degrees. If *x* zero, the result is approximately equal to the value of arctan (*y*/*x*).

If *y* > zero, the result is positive.

If *y* < zero, the result is negative.

If *y* = zero, the result is zero (if *x* > zero) or 180 degrees (if *x* < zero).

If *x* = zero, the absolute value of the result is 90 degrees.

Specific Name	Argument Type	Result Type
ATAN2D	REAL(4)	REAL(4)
DATAN2D	REAL(8)	REAL(8)
QATAN2D	REAL(16)	REAL(16)

Example

ATAN2D (2.679676, 1.0) has the value 69.53546.

ATAND

Elemental Intrinsic Function (Generic):
Produces the arctangent of x.

Syntax

```
result = ATAND (x)
```

x	(Input) Must be of type real and must be greater than or equal to zero.

Results

The result type is the same as *x* and is expressed in degrees.

Specific Name	Argument Type	Result Type
ATAND	REAL(4)	REAL(4)
DATAND	REAL(8)	REAL(8)
QATAND	REAL(16)	REAL(16)

Example

ATAND (0.0874679) has the value 4.998819.

ATANH

Elemental Intrinsic Function (Generic):
Produces the hyperbolic arctangent of x.

Syntax

```
result = ATANH (x)
```

x	(Input) Must be of type real, where | *x* | is less than or equal to 1.

Results

The result type is the same as *x*. The value lies in the range -1.0 to 1.0.

Specific Name	Argument Type	Result Type
ATANH	REAL(4)	REAL(4)
DATANH	REAL(8)	REAL(8)
QATANH	REAL(16)	REAL(16)

Example

ATANH (-0.77) has the value -1.02033.

ATANH (0.5) has the value 0.549306.

ATOMIC

OpenMP* Fortran Compiler Directive: *Ensures that a specific memory location is updated dynamically; this prevents the possibility of multiple, simultaneous writing threads.*

Syntax

```
c$OMP ATOMIC
```

c — Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

The ATOMIC directive permits optimization beyond that of the critical section around the assignment. An implementation can replace ATOMIC directives by enclosing each statement in a critical section. The critical section (or sections) must use the same unique name.

The ATOMIC directive applies only to the immediately following statement, which must have one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type
- *expr* is a scalar expression that does not reference *x*
- *intrinsic* is MAX, MIN, IAND, IOR, or IEOR
- *operator* is +, *, -, /, .AND., .OR., .EQV., or .NEQV.

All references to storage location *x* must have the same type and type parameters.

Only the loading and storing of *x* are dynamic; the evaluation of *expr* is not dynamic. To avoid race conditions (or concurrency races), all updates of the location in parallel must be protected using the ATOMIC directive, except those that are known to be free of race conditions. The function *intrinsic*, the operator *operator*, and the assignment must be the intrinsic function, operator, and assignment.

Example

The following example shows a way to avoid race conditions by using ATOMIC to protect all simultaneous updates of the location by multiple threads:

```
c$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
      DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)
c$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
      END DO
```

Since the ATOMIC directive applies only to the statement immediately following it, note that Y is *not* updated atomically.

See Also

- A to B
- OpenMP Fortran Compiler Directives

ATTRIBUTES

General Compiler Directive: *Declares properties for specified variables.*

Syntax

```
cDEC$ ATTRIBUTES att[,att]...:: object[,object]...
```

c — Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

att — Is one of the following options (or properties):

ALIAS	DEFAULT	NO_ARG_CHECK
ALIGN	DLLEXPORT	NOINLINE
ALLOCATABLE	DLLIMPORT	NOMIXED_STR_LEN_ARG
ALLOW_NULL	EXTERN	REFERENCE
ARRAY_VISUALIZER	FORCEINLINE	STDCALL
C	IGNORE_LOC	VALUE
DECORATE	INLINE	VARYING

object — Is the name of a data object or procedure.

The following table shows which ATTRIBUTES options can be used with various objects:

Option	Variable and Array Declarations	Common Block Names [1]	Subprogram Specification and EXTERNAL Statements
ALIAS	No	Yes	Yes
ALIGN	Yes	No	No
ALLOCATABLE	Yes[2]	No	No
ALLOW_NULL	Yes	No	No

Option	Variable and Array Declarations	Common Block Names [1]	Subprogram Specification and EXTERNAL Statements
ARRAY_VISUALIZER	Yes[2]	No	No
C	No	Yes	Yes
DECORATE	No	No	Yes
DEFAULT	No	Yes	Yes
DLLEXPORT	Yes[3]	Yes	Yes
DLLIMPORT	Yes	Yes	Yes
EXTERN	Yes	No	No
FORCEINLINE	No	No	Yes
IGNORE_LOC	Yes[4]	No	No
INLINE	No	No	Yes
NO_ARG_CHECK	Yes	No	Yes[5]
NOINLINE	No	No	Yes
NOMIXED_STR_LEN_ARG	No	No	Yes
REFERENCE	Yes	No	Yes
STDCALL	No	Yes	Yes
VALUE	Yes	No	No
VARYING	No	No	Yes

[1] A common block name is specified as [/]common-block-name[/]

[2] This option can only be applied to arrays.

Option	Variable and Array Declarations	Common Block Names [1]	Subprogram Specification and EXTERNAL Statements

[3]Module-level variables and arrays only.

[4]This option can only be applied to INTERFACE blocks.

[5]This option cannot be applied to EXTERNAL statements.

These options can be used in function and subroutine definitions, in type declarations, and with the INTERFACE and ENTRY statements.

Options applied to entities available through use or host association are in effect during the association. For example, consider the following:

```
MODULE MOD1  INTERFACE
    SUBROUTINE SUB1
    !DEC$ ATTRIBUTES C, ALIAS:'othername' :: NEW_SUB
    END SUBROUTINE
  END INTERFACE
  CONTAINS
    SUBROUTINE SUB2
    CALL NEW_SUB
    END SUBROUTINE
END MODULE
```

In this case, the call to NEW_SUB within SUB2 uses the C and ALIAS options specified in the interface block.

The following are ATTRIBUTES options:

- ALIAS
- ALIGN
- ALLOCATABLE
- ALLOW_NULL
- ARRAY_VISUALIZER
- C and STDCALL

- DECORATE
- DEFAULT
- DLLEXPORT and DLLIMPORT
- EXTERN
- IGNORE_LOC
- INLINE, NOINLINE, and FORCEINLINE
- NO_ARG_CHECK
- NOMIXED_STR_LEN_ARG
- REFERENCE and VALUE
- VARYING

Options C, STDCALL, REFERENCE, VALUE, and VARYING affect the calling conventions of routines:

- You can specify C, STDCALL, REFERENCE, and VARYING for an entire routine.
- You can specify VALUE and REFERENCE for individual arguments.

Examples

```
INTERFACE
   SUBROUTINE For_Sub (I)
     !DEC$ ATTRIBUTES C, ALIAS:'_For_Sub' :: For_Sub
     INTEGER I
   END SUBROUTINE For_Sub
END INTERFACE
```

You can assign more than one option to multiple variables with the same compiler directive. All assigned options apply to all specified variables. For example:

```
!DEC$ ATTRIBUTES REFERENCE, VARYING, C :: A, B, C
```

In this case, the variables `A`, `B`, and `C` are assigned the REFERENCE, VARYING, and C options. The only restriction on the number of options and variables is that the entire compiler directive must fit on one line.

The identifier of the variable or procedure that is assigned one or more options must be a simple name. It cannot include initialization or array dimensions. For example, the following is not allowed:

```
!DEC$ ATTRIBUTES C :: A(10)  ! This is illegal.
```

The following shows another example:

```
SUBROUTINE ARRAYTEST(arr)
!DEC$ ATTRIBUTES DLLEXPORT :: ARRAYTEST
   REAL(4) arr(3, 7)
   INTEGER i, j
   DO i = 1, 3
     DO j = 1, 7
       arr (i, j) = 11.0 * i + j
     END DO
   END DO
END SUBROUTINE
```

See Also

- A to B
- ATTRIBUTES ALIAS
- ATTRIBUTES ALIGN
- ATTRIBUTES ALLOCATABLE
- ATTRIBUTES ALLOW_NULL
- ATTRIBUTES ARRAY_VISUALIZER (W*32, W*64)
- ATTRIBUTES C and STDCALL
- ATTRIBUTES DECORATE
- ATTRIBUTES DEFAULT
- ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)
- ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)
- ATTRIBUTES EXTERN
- ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE
- ATTRIBUTES IGNORE_LOC
- ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE
- ATTRIBUTES MIXED_STR_LEN_ARG and NOMIXED_STR_LEN_ARG
- ATTRIBUTES NO_ARG_CHECK
- ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE
- ATTRIBUTES MIXED_STR_LEN_ARG and NOMIXED_STR_LEN_ARG
- ATTRIBUTES REFERENCE and VALUE

- ATTRIBUTES C and STDCALL
- ATTRIBUTES REFERENCE and VALUE
- ATTRIBUTES VARYING
- General Compiler Directives

Programming with Mixed Languages Overview

ATTRIBUTES ALIAS

The ATTRIBUTES directive option ALIAS specifies an alternate external name to be used when referring to external subprograms. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES ALIAS: external-name:: subprogram
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
external-name	Is a character constant delimited by apostrophes or quotation marks. The character constant is used as is; the string is not changed to uppercase, nor are blanks removed.
subprogram	Is an external subprogram.

The ALIAS option overrides the C (and STDCALL) option. If both C and ALIAS are specified for a subprogram, the subprogram is given the C calling convention, but not the C naming convention. It instead receives the name given for ALIAS, with no modifications.

ALIAS cannot be used with internal procedures, and it cannot be applied to dummy arguments.

The following example gives the subroutine happy the name "_OtherName@4" outside this scoping unit:

```
INTERFACE
  SUBROUTINE happy(i)
    !DEC$ ATTRIBUTES STDCALL, DECORATE, ALIAS:'OtherName' :: happy
    INTEGER i
  END SUBROUTINE
END INTERFACE
```

*c*DEC$ ATTRIBUTES ALIAS has the same effect as the cDEC$ ALIAS directive.

ATTRIBUTES ALIGN

The ATTRIBUTES directive option ALIGN specifies the byte alignment for a variable. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES ALIGN: n:: var
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is the number of bytes for the minimum alignment boundary. For allocatable variables, the boundary value must be a power of 2 between 1 and 16384, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on. For non-allocatable variables, the boundary value must be a power of 2 between 1 and 64 on Windows* systems, between 1 and 2**16 on Linux* systems, or between 1 and 2**12 on Mac OS* X systems.
var	Is the variable to be aligned.

ATTRIBUTES ALLOCATABLE

The ATTRIBUTES directive option ALLOCATABLE is provided for compatibility with older programs. It lets you delay allocation of storage for a particular declared entity until some point at run time when you explicitly call a routine that dynamically allocates storage for the entity. The ALLOCATABLE option takes the following form:

Syntax

```
cDEC$ ATTRIBUTES ALLOCATABLE :: entity
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
entity	Is the name of the entity that should have allocation delayed.

The recommended method for dynamically allocating storage is to use the ALLOCATABLE statement or attribute.

ATTRIBUTES ALLOW_NULL

The ATTRIBUTES directive option ALLOW_NULL enables a corresponding dummy argument to pass a NULL pointer (defined by a zero or the NULL intrinsic) by value for the argument. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES ALLOW_NULL :: arg
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
arg	Is the name of the argument.

ALLOW_NULL is only valid if the REFERENCE option is also specified; otherwise, it has no effect.

ATTRIBUTES ARRAY_VISUALIZER (W*32, W*64)

The ATTRIBUTES directive option ARRAY_VISUALIZER enables more efficient memory sharing between the application and the Intel® Array Visualizer library. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES ARRAY_VISUALIZER :: array
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
array	Is the array to be used with the Intel® Array Visualizer library.

The following example shows a way to use this directive option to improve the performance of the call:

```
real(4), allocatable :: MyArray(:, :)

!DEC$ ATTRIBUTES array_visualizer :: MyArray
```

ATTRIBUTES C and STDCALL

The ATTRIBUTES directive options C and STDCALL specify procedure calling, naming, and argument passing conventions. They take the following forms:

Syntax

```
cDEC$ ATTRIBUTES C :: object[, object] ...
```

*c*DEC$ ATTRIBUTES STDCALL :: *object*[, *object*] ...

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
object	Is the name of a data object or procedure.

On W*32 only (see Conventions, Platform labels), C and STDCALL have slightly different meanings; on all other platforms, they are interpreted as synonyms.

When applied to a subprogram, these options define the subprogram as having a specific set of calling conventions.

The following table summarizes the differences between the calling conventions:

Convention	C [1]	STDCALL [1]	Default [2]
Arguments passed by value	Yes	Yes	No
Case of external subprogram names	L*X, M*X: Lowercase W*32, W*64: Lowercase	L*X, M*X: Lowercase W*32, W*64: Lowercase	L*X, M*X: Lowercase W*32, W*64: Uppercase
L*X, M*X only:			
Trailing underscore added	No	No	Yes[3]
M*X only:			
Leading underscore added	No	No	Yes
W*32 only:			
Leading underscore added	Yes	Yes	Yes[4]
Number of argument bytes added to name	No	Yes	No
Caller stack cleanup	Yes	No	Yes

Convention	C [1]	STDCALL [1]	Default [2]
Variable number of arguments	Yes	No	Yes

[1]C and STDCALL are synonyms on Linux systems.

[2]The Intel Fortran calling convention

[3]On Linux systems, if there are one or more underscores in the external name, two trailing underscores are added; if there are no underscores, one is added.

[4]W*32 only

If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran conventions pass arguments by reference.

On IA-32 architecture, an underscore (_) is placed at the beginning of the external name of a subprogram. If STDCALL is specified, an at sign (@) followed by the number of argument bytes being passed is placed at the end of the name. For example, a subprogram named SUB1 that has three INTEGER(4) arguments and is defined with STDCALL is assigned the external name _sub1@12.

Character arguments are passed as follows:

- By default, hidden lengths are put at the end of the argument list.

  On Windows* systems using IA-32 architecture, you can get Compaq* Visual Fortran default behavior by specifying compiler option iface.
- If C or STDCALL (only) is specified:

  On all systems, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
- If C or STDCALL is specified, and REFERENCE is specified for the argument:

  On all systems, the string is passed with no length.
- If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is *not* specified for the argument, if any):

  On all systems, the string is passed with the length.

See Also

- ATTRIBUTES
- ATTRIBUTES
- ATTRIBUTES
- REFERENCE

Building Applications: Adjusting Calling Conventions in Mixed-Language Programming Overview

ATTRIBUTES DECORATE

The ATTRIBUTES directive option DECORATE specifies that the external name used in cDEC$ ALIAS or cDEC$ ATTRIBUTES ALIAS should have the prefix and postfix decorations performed on it that are associated with the calling mechanism that is in effect. These are the same decorations performed on the procedure name when ALIAS is not specified.

Syntax

The DECORATE option takes the following form:

```
cDEC$ ATTRIBUTES DECORATE :: exname
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
exname	Is an external name.

The case of the ALIAS external name is not modified.

If ALIAS is not specified, this option has no effect.

See Also

- ATTRIBUTES
- The example in the description of the ATTRIBUTES option ALIAS
- The summary of prefix and postfix decorations in the description of the ATTRIBUTES options C and STDCALL

ATTRIBUTES DEFAULT

The ATTRIBUTES directive option DEFAULT overrides certain compiler options that can affect external routine and COMMON block declarations. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES DEFAULT :: entity
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
entity	Is an external procedure or COMMON block.

It specifies that the compiler should ignore compiler options that change the default conventions for external symbol naming and argument passing for routines and COMMON blocks (such as names, assume underscore, assume 2underscores on Linux systems, and iface on Windows* systems).

This option can be combined with other ATTRIBUTES options, such as STDCALL, C, REFERENCE, ALIAS, etc. to specify properties different from the compiler defaults.

This option is useful when declaring INTERFACE blocks for external routines, since it prevents compiler options from changing calling or naming conventions.

See Also

- ATTRIBUTES
- names compiler option
- assume compiler option

ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)

The ATTRIBUTES directive options DLLEXPORT and DLLIMPORT define a dynamic-link library's (DLL) interface for processes that use them. The options can be assigned to module variables, COMMON blocks, and procedures. They take the following forms:

Syntax

```
cDEC$ ATTRIBUTES DLLEXPORT :: object[, object] ...
cDEC$ ATTRIBUTES DLLIMPORT :: object[, object] ...
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
object	Is the name of a module variable, COMMON block, or procedure. The name of a COMMON block must be enclosed in slashes.

DLLEXPORT specifies that procedures or data are being exported to other applications or DLLs. This causes the compiler to produce efficient code, eliminating the need for a module definition (.def) file to export symbols.

DLLEXPORT should be specified in the routine to which it applies.

Symbols defined in a DLL are imported by programs that use them. The program must link with the DLL import library (.lib) and use the DLLIMPORT option inside the program unit that imports the symbol. DLLIMPORT is specified in a declaration, not a definition, since you cannot define a symbol you are importing.

See Also

- ATTRIBUTES
- ATTRIBUTES

Building Applications: Creating and Using Fortran DLLs Overview, for details on working with DLL applications

ATTRIBUTES DLLEXPORT and DLLIMPORT (W*32, W*64)

The ATTRIBUTES directive options DLLEXPORT and DLLIMPORT define a dynamic-link library's (DLL) interface for processes that use them. The options can be assigned to module variables, COMMON blocks, and procedures. They take the following forms:

Syntax

```
cDEC$ ATTRIBUTES DLLEXPORT :: object[, object] ...

cDEC$ ATTRIBUTES DLLIMPORT :: object[, object] ...
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
object	Is the name of a module variable, COMMON block, or procedure. The name of a COMMON block must be enclosed in slashes.

DLLEXPORT specifies that procedures or data are being exported to other applications or DLLs. This causes the compiler to produce efficient code, eliminating the need for a module definition (.def) file to export symbols.

DLLEXPORT should be specified in the routine to which it applies.

Symbols defined in a DLL are imported by programs that use them. The program must link with the DLL import library (.lib) and use the DLLIMPORT option inside the program unit that imports the symbol. DLLIMPORT is specified in a declaration, not a definition, since you cannot define a symbol you are importing.

See Also

- ATTRIBUTES
- ATTRIBUTES

Building Applications: Creating and Using Fortran DLLs Overview, for details on working with DLL applications

ATTRIBUTES EXTERN

The ATTRIBUTES directive option EXTERN specifies that a variable is allocated in another source file. EXTERN can be used in global variable declarations, but it must not be applied to dummy arguments. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES EXTERN :: var
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
var	Is the variable to be allocated.

This option must be used when accessing variables declared in other languages.

ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE

The ATTRIBUTES directive options INLINE, NOINLINE, and FORCEINLINE can be used to control inlining decisions made by the compiler. You should place the directive option in the procedure whose inlining you want to influence.

Syntax

The INLINE option specifies that a function or subroutine can be inlined. The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size. It takes the following form:

```
cDEC$ ATTRIBUTES INLINE :: procedure
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
procedure	Is the function or subroutine that can be inlined.

The NOINLINE option disables inlining of a function. It takes the following form:

```
cDEC$ ATTRIBUTES NOINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must not be inlined.

The FORCEINLINE option specifies that a function or subroutine must be inlined unless it will cause errors. It takes the following form:

```
cDEC$ ATTRIBUTES FORCEINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must be inlined.

ATTRIBUTES IGNORE_LOC

The ATTRIBUTES directive option IGNORE_LOC enables %LOC to be stripped from an argument. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES IGNORE_LOC :: arg
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
arg	Is the name of an argument.

IGNORE_LOC is only valid if the REFERENCE option is also specified; otherwise, it has no effect.

ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE

The ATTRIBUTES directive options INLINE, NOINLINE, and FORCEINLINE can be used to control inlining decisions made by the compiler. You should place the directive option in the procedure whose inlining you want to influence.

Syntax

The INLINE option specifies that a function or subroutine can be inlined. The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size. It takes the following form:

```
cDEC$ ATTRIBUTES INLINE :: procedure
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
procedure	Is the function or subroutine that can be inlined.

The NOINLINE option disables inlining of a function. It takes the following form:

```
cDEC$ ATTRIBUTES NOINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must not be inlined.

The FORCEINLINE option specifies that a function or subroutine must be inlined unless it will cause errors. It takes the following form:

```
cDEC$ ATTRIBUTES FORCEINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must be inlined.

ATTRIBUTES MIXED_STR_LEN_ARG and NOMIXED_STR_LEN_ARG

These ATTRIBUTES directive options specify where hidden lengths for character arguments and character-valued functions should be placed. MIXED_STR_LEN_ARG specifies that hidden lengths for character arguments and character-valued functions should be placed immediately following the argument address in the argument list. NOMIXED_STR_LEN_ARG specifies that these hidden lengths should be placed in sequential order at the end of the argument list. They take the following form:

Syntax

```
cDEC$ ATTRIBUTES MIXED_STR_LEN_ARG :: args
```

```
cDEC$ ATTRIBUTES NOMIXED_STR_LEN_ARG :: args
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
args	Is a list of arguments.

The default is NOMIXED_STR_LEN_ARG. However, If you specify compiler option `/iface:CVF` or `/iface:mixed_str_len_arg` (Windows), or compiler option `-mixed-str-len-arg` (Linux and Mac OS X), the default is MIXED_STR_LEN_ARG.

See Also

- ATTRIBUTES
- ATTRIBUTES
- ATTRIBUTES

ATTRIBUTES NO_ARG_CHECK

The ATTRIBUTES directive option NO_ARG_CHECK specifies that type and shape matching rules related to explicit interfaces are to be ignored. This permits the construction of an INTERFACE block for an external procedure or a module procedure that accepts an argument of any type or shape; for example, a memory copying routine. The NO_ARG_CHECK option takes the following form:

Syntax

`cDEC$ ATTRIBUTES NO_ARG_CHECK :: object`

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
object	Is the name of an argument or procedure.

NO_ARG_CHECK can appear only in an INTERFACE block for a non-generic procedure or in a module procedure. It can be applied to an individual dummy argument name or to the routine name, in which case the option is applied to all dummy arguments in that interface.

NO_ARG_CHECK *cannot* be used for procedures with the PURE or ELEMENTAL prefix.

See Also

- ATTRIBUTES
- ATTRIBUTES

ATTRIBUTES INLINE, NOINLINE, and FORCEINLINE

The ATTRIBUTES directive options INLINE, NOINLINE, and FORCEINLINE can be used to control inlining decisions made by the compiler. You should place the directive option in the procedure whose inlining you want to influence.

Syntax

The INLINE option specifies that a function or subroutine can be inlined. The inlining can be ignored by the compiler if inline heuristics determine it may have a negative impact on performance or will cause too much of an increase in code size. It takes the following form:

```
cDEC$ ATTRIBUTES INLINE :: procedure
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
procedure	Is the function or subroutine that can be inlined.

The NOINLINE option disables inlining of a function. It takes the following form:

```
cDEC$ ATTRIBUTES NOINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must not be inlined.

The FORCEINLINE option specifies that a function or subroutine must be inlined unless it will cause errors. It takes the following form:

```
cDEC$ ATTRIBUTES FORCEINLINE :: procedure
```

c	See above.
procedure	Is the function or subroutine that must be inlined.

ATTRIBUTES MIXED_STR_LEN_ARG and NOMIXED_STR_LEN_ARG

These ATTRIBUTES directive options specify where hidden lengths for character arguments and character-valued functions should be placed. MIXED_STR_LEN_ARG specifies that hidden lengths for character arguments and character-valued functions should be placed immediately following the argument address in the argument list. NOMIXED_STR_LEN_ARG specifies that these

hidden lengths should be placed in sequential order at the end of the argument list. They take the following form:

Syntax

```
cDEC$ ATTRIBUTES MIXED_STR_LEN_ARG :: args
```

```
cDEC$ ATTRIBUTES NOMIXED_STR_LEN_ARG :: args
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
args	Is a list of arguments.

The default is NOMIXED_STR_LEN_ARG. However, If you specify compiler option `/iface:CVF` or `/iface:mixed_str_len_arg` (Windows), or compiler option `-mixed-str-len-arg` (Linux and Mac OS X), the default is MIXED_STR_LEN_ARG.

See Also

- ATTRIBUTES
- ATTRIBUTES
- ATTRIBUTES

ATTRIBUTES REFERENCE and VALUE

The ATTRIBUTES directive options REFERENCE and VALUE specify how a dummy argument is to be passed. They take the following form:

Syntax

```
cDEC$ ATTRIBUTES REFERENCE :: arg
```

```
cDEC$ ATTRIBUTES VALUE :: arg
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
arg	Is the name of a dummy argument.

REFERENCE specifies a dummy argument's memory location is to be passed instead of the argument's value.

VALUE specifies a dummy argument's value is to be passed instead of the argument's memory location.

When VALUE is specified for a dummy argument, the actual argument passed to it can be of a different type. If necessary, type conversion is performed before the subprogram is called.

When a complex (KIND=4 or KIND=8) argument is passed by value, *two* floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value.

Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value.

If REFERENCE (only) is specified for a character argument, the string is passed with no length.

If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine.

If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has *not* been specified for the argument, the string is passed with the length.

VALUE is the default if the C or STDCALL option is specified in the subprogram definition.

In the following example integer x is passed by value:

```
      SUBROUTINE Subr (x)
      INTEGER x
!DEC$ ATTRIBUTES VALUE :: x
```

See Also

- ATTRIBUTES
- ATTRIBUTES
- C and STDCALL

Building Applications: Adjusting Calling Conventions in Mixed-Language Programming Overview

ATTRIBUTES C and STDCALL

The ATTRIBUTES directive options C and STDCALL specify procedure calling, naming, and argument passing conventions. They take the following forms:

Syntax

```
cDEC$ ATTRIBUTES C :: object[, object] ...
cDEC$ ATTRIBUTES STDCALL :: object[, object] ...
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
object	Is the name of a data object or procedure.

On W*32 only (see Conventions, Platform labels), C and STDCALL have slightly different meanings; on all other platforms, they are interpreted as synonyms.

When applied to a subprogram, these options define the subprogram as having a specific set of calling conventions.

The following table summarizes the differences between the calling conventions:

Convention	**C** [1]	**STDCALL** [1]	**Default** [2]
Arguments passed by value	Yes	Yes	No
Case of external subprogram names	L*X, M*X: Lowercase W*32, W*64: Lowercase	L*X, M*X: Lowercase W*32, W*64: Lowercase	L*X, M*X: Lowercase W*32, W*64: Uppercase
L*X, M*X only:			
Trailing underscore added	No	No	Yes[3]
M*X only:			
Leading underscore added	No	No	Yes
W*32 only:			
Leading underscore added	Yes	Yes	Yes[4]
Number of argument bytes added to name	No	Yes	No
Caller stack cleanup	Yes	No	Yes

Convention	C [1]	STDCALL [1]	Default [2]
Variable number of arguments	Yes	No	Yes

[1]C and STDCALL are synonyms on Linux systems.

[2]The Intel Fortran calling convention

[3]On Linux systems, if there are one or more underscores in the external name, two trailing underscores are added; if there are no underscores, one is added.

[4]W*32 only

If C or STDCALL is specified for a subprogram, arguments (except for arrays and characters) are passed by value. Subprograms using standard Fortran conventions pass arguments by reference.

On IA-32 architecture, an underscore (_) is placed at the beginning of the external name of a subprogram. If STDCALL is specified, an at sign (@) followed by the number of argument bytes being passed is placed at the end of the name. For example, a subprogram named SUB1 that has three INTEGER(4) arguments and is defined with STDCALL is assigned the external name _sub1@12.

Character arguments are passed as follows:

- By default, hidden lengths are put at the end of the argument list.

  On Windows* systems using IA-32 architecture, you can get Compaq* Visual Fortran default behavior by specifying compiler option iface.
- If C or STDCALL (only) is specified:

  On all systems, the first character of the string is passed (and padded with zeros out to INTEGER(4) length).
- If C or STDCALL is specified, and REFERENCE is specified for the argument:

  On all systems, the string is passed with no length.
- If C or STDCALL is specified, and REFERENCE is specified for the routine (but REFERENCE is *not* specified for the argument, if any):

  On all systems, the string is passed with the length.

See Also

- ATTRIBUTES
- ATTRIBUTES
- ATTRIBUTES
- REFERENCE

Building Applications: Adjusting Calling Conventions in Mixed-Language Programming Overview

ATTRIBUTES REFERENCE and VALUE

The ATTRIBUTES directive options REFERENCE and VALUE specify how a dummy argument is to be passed. They take the following form:

Syntax

```
cDEC$ ATTRIBUTES REFERENCE :: arg
cDEC$ ATTRIBUTES VALUE :: arg
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
arg	Is the name of a dummy argument.

REFERENCE specifies a dummy argument's memory location is to be passed instead of the argument's value.

VALUE specifies a dummy argument's value is to be passed instead of the argument's memory location.

When VALUE is specified for a dummy argument, the actual argument passed to it can be of a different type. If necessary, type conversion is performed before the subprogram is called.

When a complex (KIND=4 or KIND=8) argument is passed by value, *two* floating-point arguments (one containing the real part, the other containing the imaginary part) are passed by immediate value.

Character values, substrings, assumed-size arrays, and adjustable arrays cannot be passed by value.

If REFERENCE (only) is specified for a character argument, the string is passed with no length.

If REFERENCE is specified for a character argument, and C (or STDCALL) has been specified for the routine, the string is passed with no length. This is true even if REFERENCE is also specified for the routine.

If REFERENCE and C (or STDCALL) are specified for a routine, but REFERENCE has *not* been specified for the argument, the string is passed with the length.

VALUE is the default if the C or STDCALL option is specified in the subprogram definition.

In the following example integer x is passed by value:

```
      SUBROUTINE Subr (x)
      INTEGER x
!DEC$ ATTRIBUTES VALUE :: x
```

See Also

- ATTRIBUTES
- ATTRIBUTES
- C and STDCALL

Building Applications: Adjusting Calling Conventions in Mixed-Language Programming Overview

ATTRIBUTES VARYING

The ATTRIBUTES directive option VARYING allows a variable number of calling arguments. It takes the following form:

Syntax

```
cDEC$ ATTRIBUTES VARYING :: var[, var] ...
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
var	Is the name of a variable.

Either the first argument must be a number indicating how many arguments to process, or the last argument must be a special marker (such as -1) indicating it is the final argument. The sequence of the arguments, and types and kinds must be compatible with the called procedure.

If VARYING is specified, the C option must also be specified.

See Also

- ATTRIBUTES
- ATTRIBUTES
- Syntax Rules for Compiler Directives

AUTOAddArg (W*32, W*64)

***AUTO Subroutine**: Passes an argument name and value and adds the argument to the argument list data structure.*

Module

USE IFAUTO

USE IFWINTY

Syntax

CALL AUTOAddArg (*invoke_args*,*name*,*value*[,*intent_arg*][,*type*])

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).
name	The argument's name of type CHARACTER*(*).
value	The argument's value. Must be of type INTEGER(1), INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), CHARACTER*(*), or a single dimension array of one of these types. Can also be of type VARIANT, which is defined in the IFWINTY module.
intent_arg	Indicates the intended use of the argument by the called method. Must be one of the following constants defined in the IFAUTO module: • AUTO_ARG_IN: The argument's value is read by the called method, but not written. This is the default value if *intent_arg* is not specified. • AUTO_ARG_OUT: The argument's value is written by the called method, but not read. • AUTO_ARG_INOUT: The argument's value is read and written by the called method. When the value of *intent_arg* is AUTO_ARG_OUT or AUTO_ARG_INOUT, the variable used in the *value* argument should be declared using the VOLATILE attribute. This is because the value of the variable will be changed by the subsequent call to AUTOInvoke. The compiler's global optimizations need to know that the value can change unexpectedly.

type The variant type of the argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I1	INTEGER(1)
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_VARIANT	TYPE(VARIANT)
VT_UNKNOWN	INTEGER(4)

Example

See the example in COMInitialize.

AUTOAllocateInvokeArgs (W*32, W*64)

***AUTO Function**: Allocates an argument list data structure that holds the arguments to be passed to AUTOInvoke.*

Module

USE IFAUTO

Syntax

```
result = AUTOAllocateInvokeArgs( )
```

Results

The value returned is an argument list data structure of type INTEGER(INT_PTR_KIND()).

Example

See the example in COMInitialize.

AUTODeallocateInvokeArgs (W*32, W*64)

***AUTO Subroutine**: Deallocates an argument list data structure.*

Module

USE IFAUTO

Syntax

```
CALL AUTODeallocateInvokeArgs (invoke_args)
```

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Example

See the example in COMInitialize.

AUTOGetExceptInfo (W*32, W*64)

***AUTO Subroutine**: Retrieves the exception information when a method has returned an exception status.*

Module

USE IFAUTO

Syntax

```
CALL AUTOGetExceptInfo
(invoke_args,code,source,description,h_file,h_context,scode)
```

invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).
code	An output argument that returns the error code. Must be of type INTEGER(2).
source	An output argument that returns a human-readable name of the source of the exception. Must be of type CHARACTER*(*).
description	An output argument that returns a human-readable description of the error. Must be of type CHARACTER*(*).
h_file	An output argument that returns the fully qualified path of a Help file with more information about the error. Must be of type CHARACTER*(*).
h_context	An output argument that returns the Help context of the topic within the Help file. Must be of type INTEGER(4).
scode	An output argument that returns an SCODE describing the error. Must be of type INTEGER(4).

AUTOGetProperty (W*32, W*64)

***AUTO Function**: Passes the name or identifier of the property and gets the value of the automation object's property.*

Module

USE IFAUTO

USE IFWINTY

Syntax

```
result = AUTOGetProperty (idispatch,id,value[,type])
```

idispatch The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

id The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value An output argument that returns the argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type The variant type of the requested argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_UNKNOWN	INTEGER(4)

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyByID (W*32, W*64)

***AUTO Function**: Passes the member ID of the property and gets the value of the automation object's property into the argument list's first argument.*

Module

USE IFAUTO

Syntax

`result = AUTOGetPropertyByID (idispatch,memid,invoke_args)`

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
memid	Member ID of the property. Must be of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOGetPropertyInvokeArgs (W*32, W*64)

***AUTO Function**: Passes an argument list data structure and gets the value of the automation object's property specified in the argument list's first argument.*

Module

USE IFAUTO

Syntax

`result = AUTOGetPropertyInvokeArgs (idispatch,invoke_args)`

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(INT_PTR_KIND()).

AUTOInvoke (W*32, W*64)

***AUTO Function**: Passes the name or identifier of an object's method and an argument list data structure and invokes the method with the passed arguments.*

Module

USE IFAUTO

Syntax

`result = AUTOInvoke (idispatch,id,invoke_args)`

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
id	The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

Example

See the example in COMInitialize.

AUTOMATIC

Statement and Attribute: *Controls the storage allocation of variables in subprograms (as does STATIC). Variables declared as AUTOMATIC and allocated in memory reside in the stack storage area, rather than at a static memory location.*

Syntax

The AUTOMATIC attribute can be specified in a type declaration statement or an AUTOMATIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] AUTOMATIC [, att-ls] :: v[, v] ...
```

Statement:

```
AUTOMATIC [::] v[, v] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
v	Is the name of a variable or an array specification. It can be of any type.

AUTOMATIC declarations only affect how data is allocated in storage.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

Automatic variables can reduce memory use because only the variables currently being used are allocated to memory.

Automatic variables allow possible recursion. With recursion, a subprogram can call itself (directly or indirectly), and resulting values are available upon a subsequent call or return to the subprogram. For recursion to occur, RECURSIVE must be specified in one of the following ways:

- As a keyword in a FUNCTION or SUBROUTINE statement
- As a compiler option
- As an option in an OPTIONS statement

By default, the compiler allocates local scalar variables on the stack. Other non-allocatable variables of non-recursive subprograms are allocated in static storage by default. This default can be changed through compiler options. Appropriate use of the SAVE attribute may be required if your program assumes that local variables retain their definition across subprogram calls.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE (in one of the ways mentioned above).

To override any compiler option that may affect variables, explicitly specify the variables as AUTOMATIC.

NOTE. Variables that are data-initialized, and variables in COMMON and SAVE statements are always static. This is regardless of whether a compiler option specifies recursion.

A variable cannot be specified as AUTOMATIC more than once in the same scoping unit.

If the variable is a pointer, AUTOMATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as AUTOMATIC. The following table shows these restrictions:

Variable	AUTOMATIC
Dummy argument	No
Automatic object	No
Common block item	No
Use-associated item	No
Function result	No
Component of a derived type	No

If a variable is in a module's outer scope, it *cannot* be specified as AUTOMATIC.

Example

The following example shows a type declaration statement specifying the AUTOMATIC attribute:

```
REAL, AUTOMATIC :: A, B, C
```

The following example uses an AUTOMATIC statement:

```
...
CONTAINS
 INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
   STATIC C, D, E
   ...
 END FUNCTION
...
C       In this example, all variables within the program unit
C       are saved, except for "var1" and "var3". These are
C       explicitly declared in an AUTOMATIC statement, and thus have
C       memory locations on the stack:
        SUBROUTINE DoIt (arg1, arg2)
        INTEGER(4) arg1, arg2
        INTEGER(4) var1, var2, var3, var4
        SAVE
        AUTOMATIC var1, var3
C       var2 and var4 are saved
```

See Also

- A to B
- STATIC
- SAVE
- Type declaration statements
- Compatible attributes
- RECURSIVE
- OPTIONS
- POINTER

- Modules and Module Procedures
- recursive compiler option

AUTOSetProperty (W*32, W*64)

***AUTO Function**: Passes the name or identifier of the property and a value, and sets the value of the automation object's property.*

Module

USE IFAUTO

USE IFWINTY

Syntax

```
result = AUTOSetProperty (idispatch,id,value[,type])
```

idispatch The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

id The argument's name of type CHARACTER*(*), or its member ID of type INTEGER(4).

value The argument's value. Must be of type INTEGER(2), INTEGER(4), REAL(4), REAL(8), LOGICAL(2), LOGICAL(4), CHARACTER*(*), or a single dimension array of one of these types.

type The variant type of the argument. Must be one of the following constants defined in the IFWINTY module:

VARIANT Type	Value Type
VT_I2	INTEGER(2)
VT_I4	INTEGER(4)
VT_R4	REAL(4)
VT_R8	REAL(8)
VT_CY	REAL(8)
VT_DATE	REAL(8)
VT_BSTR	CHARACTER*(*)

VARIANT Type	Value Type
VT_DISPATCH	INTEGER(4)
VT_ERROR	INTEGER(4)
VT_BOOL	LOGICAL(2)
VT_UNKNOWN	INTEGER(4)

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyByID (W*32, W*64)

***AUTO Function**: Passes the member ID of the property and sets the value of the automation object's property into the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOSetPropertyByID (idispatch,memid,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
memid	Member ID of the property. Must be of type INTEGER(4).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

AUTOSetPropertyInvokeArgs (W*32, W*64)

***AUTO Function**: Passes an argument list data structure and sets the value of the automation object's property specified in the argument list's first argument.*

Module

USE IFAUTO

Syntax

```
result = AUTOSetPropertyInvokeArgs (idispatch,invoke_args)
```

idispatch	The object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
invoke_args	The argument list data structure. Must be of type INTEGER(INT_PTR_KIND()).

Results

Returns an HRESULT describing the status of the operation. Must be of type INTEGER(4).

BACKSPACE

***Statement:** Positions a sequential file at the beginning of the preceding record, making it available for subsequent I/O processing. It takes one of the following forms:*

Syntax

```
BACKSPACE ([UNIT=]io-unit[, ERR=label] [, IOSTAT=i-var])

BACKSPACE io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The I/O unit number must specify an open file on disk or magnetic tape.

Backspacing from the current record *n* is performed by rewinding to the start of the file and then performing *n* - 1 successive READs to reach the previous record.

A BACKSPACE statement must not be specified for a file that is open for direct or append access, because *n* is not available to the Fortran I/O system.

BACKSPACE cannot be used to skip over records that have been written using list-directed or namelist formatting.

If a file is already positioned at the beginning of a file, a BACKSPACE statement has no effect.

If the file is positioned between the last record and the end-of-file record, BACKSPACE positions the file at the start of the last record.

Example

```
BACKSPACE 5

BACKSPACE (5)

BACKSPACE lunit

BACKSPACE (UNIT = lunit, ERR = 30, IOSTAT = ios)
```

The following statement repositions the file connected to I/O unit 4 back to the preceding record:

```
BACKSPACE 4
```

Consider the following statement:

```
BACKSPACE (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 back to the preceding record. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

See Also

- A to B
- REWIND
- ENDFILE
- Data Transfer I/O Statements
- Branch Specifiers

BADDRESS

Inquiry Intrinsic Function (Generic): *Returns the address of x. This function cannot be passed as an actual argument. This function can also be specified as IADDR.*

Syntax

```
result = BADDRESS (x)
```

x	Is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of a statement function. If it is a pointer, it must be defined and associated with a target.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

Example

```
PROGRAM batest
  INTEGER X(5), I
  DO I=1, 5
    PRINT *, BADDRESS(X(I))
  END DO
END
```

BARRIER

OpenMP* Fortran Compiler Directive: *Synchronizes all the threads in a team. It causes each thread to wait until all of the other threads in the team have reached the barrier.*

Syntax

```
c$OMP BARRIER
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

The BARRIER directive must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

The directive binding rules call for a BARRIER directive to bind to the closest enclosing PARALLEL directive. In the following example, the BARRIER directive ensures that all threads have executed the first loop and that it is safe to execute the second loop:

```
c$OMP PARALLEL
c$OMP DO PRIVATE(i)
      DO i = 1, 100
        b(i) = i
      END DO
c$OMP BARRIER
c$OMP DO PRIVATE(i)
      DO i = 1, 100
        a(i) = b(101-i)
      END DO
c$OMP END PARALLEL
```

See Also

- A to B
- OpenMP Fortran Compiler Directives
- Nesting and Binding Rules

BEEPQQ

Portability Subroutine: *Sounds the speaker at the specified frequency for the specified duration in milliseconds.*

Module

USE IFPORT

Syntax

```
CALL BEEPQQ (frequency,duration)
```

frequency	(Input) INTEGER(4). Frequency of the tone in Hz.
duration	(Input) INTEGER(4). Length of the beep in milliseconds.

BEEPQQ does not return until the sound terminates.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) frequency, duration
frequency = 4000
duration  = 1000
CALL BEEPQQ(frequency, duration)
```

See Also

- A to B
- SLEEPQQ

BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

Portability Functions: *Compute the single-precision values of Bessel functions of the first and second kinds.*

Module

USE IFPORT

Syntax

```
result = BESJ0 (value)
result = BESJ1 (value)
result = BESJN (n, value)
result = BESY0 (posvalue)
```

```
result = BESY1 (posvalue)
result = BESYN (n, value)
```

value	(Input) REAL(4). Independent variable for a Bessel function.
n	(Input) INTEGER(4). Specifies the order of the selected Bessel function computation.
posvalue	(Input) REAL(4). Independent variable for a Bessel function. Must be greater than or equal to zero.

Results

BESJ0, BESJ1, and BESJN return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

BESY0, BESY1, and BESYN return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause BESY0, BESY1, and BESYN to return QNAN.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- A to B
- DBESJ0, DBESJ1, DBESJN

BIC, BIS

Portability Subroutines: *Perform a bit-level set and clear for integers.*

Module

USE IFPORT

Syntax

```
CALL BIC (bitnum, target)
```

CALL BIS (*bitnum*, *target*)

bitnum	(Input) INTEGER(4). Bit number to set. Must be in the range 0 (least significant bit) to 31 (most significant bit) if *target* is INTEGER(4). If *target* is INTEGER(8), *bitnum* must be in range 0 to 63.
target	(Input) INTEGER(4) or INTEGER(8). Variable whose bit is to be set.

BIC sets bit *bitnum* of *target* to 0; BIS sets bit *bitnum* to 1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

Consider the following:

```
USE IFPORT

integer(4) bitnum, target_i4

integer(8) target_i8

target_i4 = Z'AAAA'

bitnum = 1

call BIC(bitnum, target_i4)

target_i8 = Z'FFFFFFFF00000000'

bitnum = 40

call BIC(bitnum, target_i8)

bitnum = 0

call BIS(bitnum, target_i4)
```

```
bitnum = 1

call BIS(bitnum, target_i8)

print '(" integer*4 result ",Z)', target_i4

print '(" integer*8 result ",Z)', target_i8

end
```

See Also

- A to B
- BIT

BIND

Statement and Attribute: *Specifies that an object is interoperable with C and has external linkage.*

Syntax

The BIND attribute can be specified in a type declaration statement or a BIND statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] BIND (C [, NAME=*ext-name*]) [*att-ls*,] :: *object*

Statement:

BIND (C [, NAME=*ext-name*]) [::] *object*

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ext-name	Is a character scalar initialization expression that can be used to construct the external name.
object	Is the name of a variable, common block, or procedure.

Description

If a common block is specified in a BIND statement, it must be specified with the same binding label in each scoping unit in which it is declared.

For variables and common blocks, BIND also implies the SAVE attribute, which may be explicitly confirmed with SAVE.

A variable given the BIND attribute (or declared in a BIND statement) must appear in the specification part of a module. You cannot specify BIND for a subroutine local variable or a variable in a main program.

The BIND attribute is similar to directive !DEC$ ATTRIBUTES C as follows:

- The compiler applies the same naming rules, that is, names are lowercase (unless NAME= specifies otherwise).
- The compiler applies the appropriate platform decoration, such as a leading underscore.

However, procedure argument passing differs. When BIND is specified, procedure arguments are passed by reference unless the VALUE attribute is also specified.

The BIND attribute can be used in a SUBROUTINE or FUNCTION declaration.

Example

The following example shows the BIND attribute used in a type declaration statement, a statement, and a SUBROUTINE statement.

```
INTEGER, BIND(C) :: SOMEVAR

BIND(C,NAME='SharedCommon') :: /SHAREDCOMMON/

INTERFACE

     SUBROUTINE FOOBAR, BIND(C, NAME='FooBar')
     END SUBROUTINE
```

See Also

- A to B
- Modules and Module Procedures
- Type Declarations
- Compatible attributes
- Pointer Assignments
- FUNCTION
- SUBROUTINE

BIT

Portability Function: *Performs a bit-level test for integers.*

Module

USE IFPORT

Syntax

`result = BIT (bitnum, source)`

bitnum	(Input) INTEGER(4). Bit number to test. Must be in the range 0 (least significant bit) to 31 (most significant bit).
source	(Input) INTEGER(4) or INTEGER(8). Variable being tested.

Results

The result type is logical. It is .TRUE. if bit *bitnum* of *source* is 1; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- A to B
- BIC, BIS

BIT_SIZE

Inquiry Intrinsic Function (Generic): *Returns the number of bits in an integer type.*

Syntax

`result = BIT_SIZE (i)`

i	(Input) Must be of type integer or of type logical (which is treated as an integer).

Results

The result is a scalar integer with the same kind parameter as *i*. The result value is the number of bits (*s*) defined by the bit model for integers with the kind parameter of the argument. For information on the bit model, see Model for Bit Data.

Example

BIT_SIZE (1_2) has the value 16 because the KIND=2 integer type contains 16 bits.

See Also

- A to B
- BTEST
- IBCLR
- IBITS
- IBSET

BLOCK DATA

Statement: *Identifies a block-data program unit, which provides initial values for nonpointer variables in named common blocks.*

Syntax

```
BLOCK DATA [name]
    [specification-part]
BLOCK DATA [BLOCK DATA [name]]
```

name Is the name of the block data program unit.

specification-part Is one or more of the following statements:

COMMON	INTRINSIC	STATIC
DATA	PARAMETER	TARGET
Derived-type definition	POINTER	Type declaration [2]
DIMENSION	RECORD[1]	USE [3]

EQUIVALENCE	Record structure declaration[1]
IMPLICIT	SAVE

[1] For more information, see RECORD statement and record structure declarations.

[2] Can only contain attributes: DIMENSION, INTRINSIC, PARAMETER, POINTER, SAVE, STATIC,or TARGET.

[3] Allows access to only named constants.

Description

A block data program unit need not be named, but there can only be one unnamed block data program unit in an executable program.

If a name follows the END statement, it must be the same as the name specified in the BLOCK DATA statement.

An interface block must not appear in a block data program unit and a block data program unit must not contain any executable statements.

If a DATA statement initializes any variable in a named common block, the block data program unit must have a complete set of specification statements establishing the common block. However, all of the variables in the block do not have to be initialized.

A block data program unit can establish and define initial values for more than one common block, but a given common block can appear in only one block data program unit in an executable program.

The name of a block data program unit can appear in the EXTERNAL statement of a different program unit to force a search of object libraries for the block data program unit at link time.

Example

The following shows a block data program unit:

```
BLOCK DATA BLKDAT
  INTEGER S,X
  LOGICAL T,W
  DOUBLE PRECISION U
  DIMENSION R(3)
  COMMON /AREA1/R,S,U,T /AREA2/W,X,Y
  DATA R/1.0,2*2.0/, T/.FALSE./, U/0.214537D-7/, W/.TRUE./, Y/3.5/
END
```

The following shows another example:

```
C       Main Program
        CHARACTER(LEN=10) LakeType
        REAL X(10), Y(4)
        COMMON/Lakes/a,b,c,d,e,family/Blk2/x,y
        ...
C       The following block-data subprogram initializes
C       the named common block /Lakes/:
C
        BLOCK DATA InitLakes
        COMMON /Lakes/ erie, huron, michigan, ontario,
       +        superior, fname
        DATA erie, huron, michigan, ontario, superior /1, 2, 3, 4, 5/
        CHARACTER(LEN=10) fname/'GreatLakes'/
        END
```

See Also

- A to B
- COMMON

- DATA
- EXTERNAL
- Program Units and Procedures

BSEARCHQQ

Portability Function: *Performs a binary search of a sorted one-dimensional array for a specified element. The array elements cannot be derived types or structures.*

Module

USE IFPORT

Syntax

`result = BSEARCHQQ (adrkey,adrarray,length,size)`

adrkey (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Address of the variable containing the element to be found (returned by LOC).

adrarray (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Address of the array (returned by LOC).

length (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Number of elements in the array.

size (Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in `IFPORT.F90`, specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$INTEGER8	INTEGER(8) or equivalent
SRT$REAL4	REAL(4) or equivalent

Constant	Type of array
SRT$REAL8	REAL(8) or equivalent
SRT$REAL16	REAL(16) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

Results

The result type is INTEGER(4). It is an array index of the matched entry, or 0 if the entry is not found.

The array must be sorted in ascending order before being searched.

CAUTION. The location of the array and the element to be found must both be passed by address using the LOC function. This defeats Fortran type checking, so you must make certain that the *length* and *size* arguments are correct, and that *size* is the same for the element to be found and the array searched.

If you pass invalid arguments, BSEARCHQQ attempts to search random parts of memory. If the memory it attempts to search is not allocated to the current process, the program is halted, and you receive a General Protection Violation message.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) array(10), length
INTEGER(4) result, target
length = SIZE(array)
...
result = BSEARCHQQ(LOC(target),LOC(array),length,SRT$INTEGER4)
```

See Also

- A to B
- SORTQQ
- LOC

BTEST

Elemental Intrinsic Function (Generic): *Tests a bit of an integer argument.*

Syntax

`result = BTEST (i,pos)`

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.

Results

The result type is default logical.

The result is true if bit *pos* of *i* has the value 1. The result is false if *pos* has the value zero. For more information, see Bit Functions.

For information on the model for the interpretation of an integer value as a sequence of bits, see Model for Bit Data.

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument Type	Result Type
BBTEST	INTEGER(1)	LOGICAL(1)
BITEST[1]	INTEGER(2)	LOGICAL(2)
BTEST [2]	INTEGER(4)	LOGICAL(4)
BKTEST	INTEGER(8)	LOGICAL(8)

[1] Or HTEST

Specific Name	Argument Type	Result Type
[2]Or BJTEST		

Example

BTEST (9, 3) has the value true.

If A has the value

```
[ 1  2 ]
[ 3  4 ],
```

the value of BTEST (A, 2) is

```
[ false  false ]
[ false   true ]
```

and the value of BTEST (2, A) is

```
[ true   false ]
[ false  false ].
```

The following shows more examples:

Function reference	Value of *i*	Result
BTEST (*i*,2)	00011100 01111000	.FALSE.
BTEST (*i*,3)	00011100 01111000	.TRUE.

The following shows another example:

```
INTEGER(1) i(2)
LOGICAL result(2)
i(1) = 2#10101010
i(2) = 2#01010101
result = BTEST(i, (/3,2/))   ! returns (.TRUE.,.TRUE.)
write(*,*) result
```

See Also

- A to B

- IBCLR
- IBSET
- IBCHNG
- IOR
- IEOR
- IAND

BYTE

Statement: *Specifies the BYTE data type, which is equivalent to INTEGER(1).*

Example

```
BYTE count, matrix(4, 4) / 4*1, 4*2, 4(4), 4*8 /
BYTE num  / 10 /
```

See Also

- A to B
- INTEGER
- Integer Data Types

C to D

C_ASSOCIATED

Intrinsic Module Inquiry function (Generic): *Indicates the association status of one argument, or whether two arguments are associated with the same entity.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

```
result = C_ASSOCIATED(c_ptr_1[,  c_ptr_2])
```

c_ptr_1	(Input) Is a scalar of derived type C_PTR or C_FUNPTR.
c_ptr_2	(Optional; input) Is a scalar of the same type as *c_ptr_1*.

Results

The result is a scalar of type default logical. The result value is one of the following:

- If only *c_ptr_1* is specified, the result is false if *c_ptr_1* is a C null pointer; otherwise, the result is true.
- If *c_ptr_2* is specified, the result is false if *c_ptr_1* is a C null pointer. The result is true if *c_ptr_1* is equal to *c_ptr_2*; otherwise, the result is false.

See Also

- C to D
- Intrinsic Modules
- ISO_C_BINDING Module

C_F_POINTER

Intrinsic Module Subroutine: *Associates a pointer with the target of a C pointer and specifies its shape.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

```
CALL C_F_POINTER(cptr, fptr [,shape])
```

cptr	(Input) Is a scalar of derived type C_PTR. Its value is the C address of an interoperable data entity, or the result of a reference to function C_LOC with a noninteroperable argument. If the value of *cptr* is the C address of a Fortran variable, it must have the TARGET attribute.
fptr	(Output) Is a pointer. If it is an array, *shape* must be specified.
shape	(Optional, input) Must be of type integer and rank one. Its size equals the rank of *fptr*.

If the value of *cptr* is the C address of an interoperable data entity, *fptr* must be a pointer with type and type parameters interoperable with the type of the entity. In this case, *fptr* becomes pointer-associated with the target of *cptr*.

If *fptr* is an array, it has the shape specified by *shape* and each lower bound is 1.

If the value of *cptr* is the result of a reference to C_LOC with a noninteroperable argument *x*, the following rules apply:

- C_LOC argument *x* (or its target) must not have been deallocated or have become undefined due to the execution of a RETURN or END statement since the reference to C_LOC.
- *fptr* is a scalar pointer with the same type and type parameters as *x*. *fptr* becomes pointer-associated with *x*, or it becomes pointer-associated with its target if *x* is a pointer.

See Also

- C to D
- Intrinsic Modules
- ISO_C_BINDING Module
- C_LOC

C_F_PROCPOINTER

Intrinsic Module Subroutine: *Associates a Fortran pointer of type INTEGER with the target of a C function pointer.*

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

```
CALL C_F_POINTER(cptr, fptr)
```

cptr	(Input) Is a scalar of derived type C_FUNPTR. Its value is the C address of a procedure that is interoperable.
fptr	(Output) Is a Fortran pointer of type INTEGER. It becomes pointer-associated with the target of *cptr*.

See Also

- C to D
- Intrinsic Modules
- ISO_C_BINDING Module

C_FUNLOC

Intrinsic Module Inquiry function (Generic):
Returns the C address of a function pointer.

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_FUNLOC(x)`

x	(Input) Is an interoperable procedure or a Fortran pointer of type INTEGER associated with an interoperable procedure.

Results

The result is a scalar of derived type C_FUNPTR. The result value represents the C address of the argument.

See Also

- C to D
- Intrinsic Modules
- ISO_C_BINDING Module

C_LOC

Intrinsic Module Inquiry function (Generic):
Returns the C address of an argument.

Module

USE, INTRINSIC :: ISO_C_BINDING

Syntax

`result = C_LOC(x)`

x	(Input) Is one of the following: • An interoperable variable that has the TARGET attribute • An interoperable, allocatable, variable that is allocated, has the TARGET attribute, and is not an array of size zero

- An associated, interoperable scalar pointer
- A scalar that has no length type parameters and is one of the following:
  - A nonallocatable, nonpointer variable that has the TARGET attribute
  - An allocatable variable that is allocated and has the TARGET attribute
  - An associated pointer

Results

The result is a scalar of derived type C_PTR. The result value represents the C address of the argument.

The result is a value that can be used as an actual CPTR argument in a call to procedure C_F_POINTER where *fptr* has attributes that allow the pointer assignment *fptr*=>*x*. Such a call to C_F_POINTER has the effect of the pointer assignment *fptr*=>*x*.

If *x* is a scalar, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of *x* and the pointer assignment CPTR%PX=>*x* were executed.

If *x* is an array, the result is determined as if C_PTR were a derived type containing a scalar pointer component PX of the type and type parameters of *x* and the pointer assignment CPTR%PX to the first element of *x* were executed.

See Also

- C to D
- Intrinsic Modules
- ISO_C_BINDING Module
- C_F_POINTER

CACHESIZE

Inquiry Intrinsic Function (Generic): *Returns the size of a level of the memory cache.*

Syntax

```
result = CACHESIZE (n)
```

n	(Input) Must be scalar and of type integer.

Results

The result type is the same as *n*. The result value is the number of kilobytes in the level *n* memory cache.

n = 1 specifies the first level cache; *n* = 2 specifies the second level cache; etc. If cache level *n* does not exist, the result value is 0.

Example

CACHESIZE(1) returns 16 for a processor with a 16KB first level memory cache.

CALL

Statement: *Transfers control to a subroutine subprogram.*

Syntax

```
CALL sub[( [a-arg[,a-arg]...] )]
```

sub	Is the name of the subroutine subprogram or other external procedure, or a dummy argument associated with a subroutine subprogram or other external procedure.
a-arg	Is an actual argument optionally preceded by [keyword=], where *keyword* is the name of a dummy argument in the explicit interface for the subroutine. The keyword is assigned a value when the procedure is invoked. Each actual argument must be a variable, an expression, the name of a procedure, or an alternate return specifier. (It must not be the name of an internal procedure, statement function, or the generic name of a procedure.)

An alternate return specifier is an asterisk (*), or ampersand (&) followed by the label of an executable branch target statement in the same scoping unit as the CALL statement. (An alternate return is an obsolescent feature in Fortran 95 and Fortran 90.)

Description

When the CALL statement is executed, any expressions in the actual argument list are evaluated, then control is passed to the first executable statement or construct in the subroutine. When the subroutine finishes executing, control returns to the next executable statement following the CALL statement, or to a statement identified by an alternate return label (if any).

If an argument list appears, each actual argument is associated with the corresponding dummy argument by its position in the argument list or by the name of its keyword. The arguments must agree in type and kind parameters.

If positional arguments and argument keywords are specified, the argument keywords must appear last in the actual argument list.

If a dummy argument is optional, the actual argument can be omitted.

An actual argument associated with a dummy procedure must be the specific name of a procedure, or be another dummy procedure. Certain specific intrinsic function names must not be used as actual arguments (see Functions Not Allowed as Actual Arguments).

The procedure invoked by the CALL statement must be a subroutine subprogram and not a function. Calling a function as if it were a subroutine can cause unpredictable results.

Example

The following example shows valid CALL statements:

```
CALL CURVE(BASE,3.14159+X,Y,LIMIT,R(LT+2))

CALL PNTOUT(A,N,'ABCD')

CALL EXIT

CALL MULT(A,B,*10,*20,C)       ! The asterisks and ampersands denote

CALL SUBA(X,&30,&50,Y)         !     alternate returns
```

The following example shows a subroutine with argument keywords:

```
PROGRAM KEYWORD_EXAMPLE
  INTERFACE
    SUBROUTINE TEST_C(I, L, J, KYWD2, D, F, KYWD1)
    INTEGER I, L(20), J, KYWD1
    REAL, OPTIONAL :: D, F
    COMPLEX KYWD2
    ...
    END SUBROUTINE TEST_C
  END INTERFACE
  INTEGER I, J, K
  INTEGER L(20)
  COMPLEX Z1
  CALL TEST_C(I, L, J, KYWD1 = K, KYWD2 = Z1)
  ...
```

The first three actual arguments are associated with their corresponding dummy arguments by position. The argument keywords are associated by keyword name, so they can appear in any order.

Note that the interface to subroutine TEST has two optional arguments that have been omitted in the CALL statement.

The following shows another example of a subroutine call with argument keywords:

```
CALL TEST(X, Y, N, EQUALITIES = Q, XSTART = X0)
```

The first three arguments are associated by position.

The following shows another example:

```
!Variations on a subroutine call
      REAL S,T,X
      INTRINSIC NINT
      S=1.5
      T=2.5
      X=14.7
      !This calls SUB1 using keywords. NINT is an intrinsic function.
      CALL SUB1(B=X,C=S*T,FUNC=NINT,A=4.0)
!Here is the same call using an implicit reference
      CALL SUB1(4.0,X,S*T,NINT)
      CONTAINS
        SUBROUTINE sub1(a,b,c,func)
        INTEGER func
        REAL a,b,c
        PRINT *, a,b,c, func(b)
        END SUBROUTINE
      END
```

See Also

- C to D
- SUBROUTINE
- CONTAINS
- RECURSIVE
- USE
- Program Units and Procedures

CASE

Statement: *Marks the beginning of a CASE construct. A CASE construct conditionally executes one block of constructs or statements depending on the value of a scalar expression in a SELECT CASE statement.*

Syntax

```
[name:] SELECT CASE (expr)
[CASE (case-value  [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

name	Is the name of the CASE construct.
expr	Is a scalar expression of type integer, logical, or character (enclosed in parentheses). Evaluation of this expression results in a value called the *case index*.

case-value	Is one or more scalar integer, logical, or character initialization expressions enclosed in parentheses. Each *case-value* must be of the same type and kind parameter as *expr*. If the type is character, *case-value* and *expr* can be of different lengths, but their kind parameter must be the same.
	Integer and character expressions can be expressed as a range of case values, taking one of the following forms:
	`low:high`
	`low:`
	`:high`
	Case values must not overlap.

block	Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified in a SELECT CASE statement, the same name must appear in the corresponding END SELECT statement. The same construct name can optionally appear in any CASE statement in the construct. The same construct name must not be used for different named constructs in the same scoping unit.

The case expression (*expr*) is evaluated first. The resulting case index is compared to the case values to find a matching value (there can only be one). When a match occurs, the block following the matching case value is executed and the construct terminates.

The following rules determine whether a match occurs:

- When the case value is a single value (no colon appears), a match occurs as follows:

Data Type	A Match Occurs If:
Logical	case-index .EQV. case-value
Integer or Character	case-index = = case-value

- When the case value is a range of values (a colon appears), a match depends on the range specified, as follows:

Range	A Match Occurs If:
low **:**	case-index >= low
: high	case-index <= high
low **:** high	low <= case-index <= high

The following are all valid case values:

```
CASE (1, 4, 7, 11:14, 22)      ! Individual values as specified:
                               !     1, 4, 7, 11, 12, 13, 14, 22
CASE (:-1)                     ! All values less than zero
CASE (0)                       ! Only zero
CASE (1:)                      ! All values above zero
```

If no match occurs but a CASE DEFAULT statement is present, the block following that statement is executed and the construct terminates.

If no match occurs and no CASE DEFAULT statement is present, no block is executed, the construct terminates, and control passes to the next executable statement or construct following the END SELECT statement.

The following figure shows the flow of control in a **CASE** construct:

2436

Figure 47: Flow of Control in CASE Constructs

You cannot use branching statements to transfer control to a CASE statement. However, branching to a SELECT CASE statement is allowed. Branching to the END SELECT statement is allowed only from within the CASE construct.

Example

The following are examples of CASE constructs:

```
INTEGER FUNCTION STATUS_CODE (I)
  INTEGER I
  CHECK_STATUS: SELECT CASE (I)
  CASE (:-1)
    STATUS_CODE = -1
  CASE (0)
    STATUS_CODE = 0
  CASE (1:)
    STATUS_CODE = 1
  END SELECT CHECK_STATUS
END FUNCTION STATUS_CODE
SELECT CASE (J)
CASE (1, 3:7, 9)    ! Values: 1, 3, 4, 5, 6, 7, 9
  CALL SUB_A
CASE DEFAULT
  CALL SUB_B
END SELECT
```

The following three examples are equivalent:

```
1. SELECT CASE (ITEST .EQ. 1)
   CASE (.TRUE.)
     CALL SUB1 ()
   CASE (.FALSE.)
     CALL SUB2 ()
   END SELECT
2. SELECT CASE (ITEST)
   CASE DEFAULT
     CALL SUB2 ()
   CASE (1)
     CALL SUB1 ()
   END SELECT
3. IF (ITEST .EQ. 1) THEN
     CALL SUB1 ()
   ELSE
     CALL SUB2 ()
   END IF
```

The following shows another example:

```
CHARACTER*1 cmdchar
GET_ANSWER: SELECT CASE (cmdchar)
CASE ('0')
   WRITE (*, *) "Must retrieve one to nine files"
CASE ('1':'9')
   CALL RetrieveNumFiles (cmdchar)
CASE ('A', 'a')
   CALL AddEntry
CASE ('D', 'd')
   CALL DeleteEntry
CASE ('H', 'h')
   CALL Help
CASE DEFAULT
   WRITE (*, *) "Command not recognized; please use H for help"
END SELECT GET_ANSWER
```

See Also

- C to D
- Execution Control

CDFLOAT

Portability Function: *Converts a COMPLEX(4) argument to double-precision real type.*

Module

USE IFPORT

Syntax

`result = CDFLOAT (input)`

input (Input) COMPLEX(4). The value to be converted.

Results

The result type is REAL(8).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

CEILING

Elemental Intrinsic Function (Generic):
Returns the smallest integer greater than or equal to its argument.

Syntax

```
result = CEILING (a[,kind])
```

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the smallest integer greater than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Example

CEILING (4.8) has the value 5.

CEILING (-2.55) has the value -2.0.

The following shows another example:

```
INTEGER I, IARRAY(2)

I = CEILING(8.01) ! returns 9

I = CEILING(-8.01) ! returns -8

IARRAY = CEILING((/8.01,-5.6/)) ! returns (9, -5)
```

See Also

- C to D
- FLOOR

CHANGEDIRQQ

Portability Function: *Makes the specified directory the current, default directory.*

Module

USE IFPORT

Syntax

`result = CHANGEDIRQQ (dir)`

dir (Input) Character*(*). Directory to be made the current directory.

Results

The result type is LOGICAL(4). It is .TRUE. if successful; otherwise, .FALSE..

If you do not specify a drive in the *dir* string, the named directory on the current drive becomes the current directory. If you specify a drive in *dir*, the named directory on the specified drive becomes the current directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

LOGICAL(4) status

status = CHANGEDIRQQ('d:\fps90\bin\bessel')
```

See Also

- C to D
- GETDRIVEDIRQQ
- MAKEDIRQQ
- DELDIRQQ
- CHANGEDRIVEQQ

CHANGEDRIVEQQ

Portability Function: *Makes the specified drive the current, default drive.*

Module

USE IFPORT

Syntax

`result = CHANGEDRIVEQQ (drive)`

drive (Input) Character*(*). String beginning with the drive letter.

Results

The result type is LOGICAL(4). On Windows* systems, the result is .TRUE. if successful; otherwise, .FALSE. On Linux* and Mac OS* X systems, the result is always .FALSE..

Because drives are identified by a single alphabetic character, CHANGEDRIVEQQ examines only the first character of *drive*. The drive letter can be uppercase or lowercase.

CHANGEDRIVEQQ changes only the current drive. The current directory on the specified drive becomes the new current directory. If no current directory has been established on that drive, the root directory of the specified drive becomes the new current directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

LOGICAL(4) status

status = CHANGEDRIVEQQ('d')
```

See Also

- C to D
- GETDRIVESQQ
- GETDRIVESIZEQQ
- GETDRIVEDIRQQ
- CHANGEDIRQQ

CHAR

Elemental Intrinsic Function (Generic):
Returns the character in the specified position of the processor's character set. It is the inverse of the function ICHAR.

Syntax

```
result = CHAR (i[,kind])
```

i	(Input) Must be of type integer with a value in the range 0 to n - 1, where *n* is the number of characters in the processor's character set.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is of type character with length 1. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default character. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is the character in position *i* of the processor's character set. ICHAR(CHAR (*i*, *kind*(*c*))) has the value I for 0 to n - 1 and CHAR(ICHAR(*c*), *kind*(*c*)) has the value *c* for any character *c* capable of representation in the processor.

Specific Name	Argument Type	Result Type
	INTEGER(1)	CHARACTER
	INTEGER(2)	CHARACTER
CHAR [1]	INTEGER(4)	CHARACTER
	INTEGER(8)	CHARACTER

[1]This specific function cannot be passed as an actual argument.

Example

CHAR (76) has the value `'L'`.

CHAR (94) has the value `'^'`.

See Also

- C to D
- ACHAR
- IACHAR
- ICHAR
- Character and Key Code Charts

CHARACTER

Statement: *Specifies the CHARACTER data type.*

Syntax

```
CHARACTER

CHARACTER([KIND=] n)

CHARACTER*len
```

n	Is kind 1.
len	Is a string length (not a kind). For more information, see Declaration Statements for Character Types. If no kind type parameter is specified, the kind of the constant is default character.

Example

```
C
C   Length of wt and vs is 10, city is 80, and ch is 1
C
    CHARACTER wt*10, city*80, ch
    CHARACTER (LEN = 10), PRIVATE :: vs
    CHARACTER*(*) arg !declares a dummy argument
C   name and plume are ten-element character arrays
C   of length 20
    CHARACTER name(10)*20
    CHARACTER(len=20), dimension(10):: plume
C
C   Length of susan, patty, and dotty are 2, alice is 12,
C   jane is a 79-member array of length 2
C
    CHARACTER(2) susan, patty, alice*12, dotty, jane(79)
```

See Also

- C to D
- Character Data Type
- Character Constants
- Character Substrings
- C Strings
- Declaration Statements for Character Types

CHDIR

Portability Function: *Changes the default directory.*

Module

USE IFPORT

Syntax

```
result = CHDIR(dir_name)
```

dir_name	(Input) Character*(*). Name of a directory to become the default directory.

Results

The result type is INTEGER(4). It returns zero if the directory was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The named directory does not exist.
- ENOTDIR: The *dir_name* parameter is not a directory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
      use ifport
      integer(4) istatus, enoent, enotdir
      character(255)  newdir
      character(300)  prompt, errmsg
      prompt = 'Please enter directory name: '
10    write(*,*)  TRIM(prompt)
      read *, newdir
      ISTATUS = CHDIR(newdir)
      select case (istatus)
        case (2)  ! ENOENT
          errmsg = 'The directory '//TRIM(newdir)//' does not exist'
        case (20) ! ENOTDIR
          errmsg = TRIM(newdir)//' is not a directory'
        case (0) ! NO error
          goto 40
        case default
          write (errmsg,*) 'Error with code ', istatus
       end select
       write(*,*)  TRIM(errmsg)
       goto 10
40     write(*,*) 'Default directory successfully changed.'
end
```

See Also

- C to D
- CHANGEDIRQQ

CHMOD

Portability Function: *Changes the access mode of a file.*

Module

USE IFPORT

Syntax

```
result = CHMOD (name,mode)
```

name (Input) Character*(*). Name of the file whose access mode is to be changed. Must have a single path.

mode (Input) Character*(*). File permission: either Read, Write, or Execute. The *mode* parameter can be either symbolic or absolute. An absolute mode is specified with an octal number, consisting of any combination of the following permission bits ORed together:

Permission bit	Description	Action
4000	Set user ID on execution	W*32, W*64: Ignored; never true L*X, M*X: Settable
2000	Set group ID on execution	W*32, W*64: Ignored; never true L*X, M*X: Settable
1000	Sticky bit	W*32, W*64: Ignored; never true L*X, M*X: Settable
0400	Read by owner	W*32, W*64: Ignored; always true L*X, M*X: Settable
0200	Write by owner	Settable

Permission bit	Description	Action
0100	Execute by owner	W*32, W*64: Ignored; based on file name extension L*X, M*X: Settable
0040, 0020, 0010	Read, Write, Execute by group	W*32, W*64: Ignored; assumes owner permissions L*X, M*X: Settable
0004, 0002, 0001	Read, Write, Execute by others	W*32, W*64: Ignored; assumes owner permissions L*X, M*X: Settable

The following regular expression represents a symbolic mode:

```
[ugoa]*[+-=] [rwxXst]*
```

"[ugoa]*" is ignored on Windows* systems. On Linux* and Mac OS* X systems, a combination of the letters "ugoa" control which users' access to the file will be changed:

u	The user who owns the file
g	Other users in the group that owns the file
o	Other users not in the group that owns the file
a	All users

"[+ - =]" indicates the operation to carry out:

+	Add the permission
-	Remove the permission
=	Absolutely set the permission

"[rwxXst]*" indicates the permission to add, subtract, or set. On Windows systems, only "w" is significant and affects write permission; all other letters are ignored. On Linux and Mac OS X systems, all letters are significant.

Results

The result type is INTEGER(4). It is zero if the mode was changed successfully; otherwise, an error code. Possible error codes are:

- ENOENT: The specified file was not found.
- EINVAL: The mode argument is invalid.
- EPERM: Permission denied; the file's mode cannot be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

integer(4) I,Istatus

I = ACCESS ("DATAFILE.TXT", "w")

if (i) then

   ISTATUS = CHMOD ("datafile.txt", "[+w]")

end if

I = ACCESS ("DATAFILE.TXT","w")

print *, i
```

See Also

- C to D
- SETFILEACCESSQQ

CLEARSCREEN (W*32, W*64)

Graphics Subroutine: *Erases the target area and fills it with the current background color.*

Module

USE IFQWIN

Syntax

```
CALL CLEARSCREEN (area)
```

area (Input) INTEGER(4). Identifies the target area. Must be one of the following symbolic constants (defined in IFQWIN.F90):

- $GCLEARSCREEN - Clears the entire screen.
- $GVIEWPORT - Clears only the current viewport.
- $GWINDOW - Clears only the current text window (set with SETTEXTWINDOW).

All pixels in the target area are set to the color specified with SETBKCOLORRGB. The default color is black.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN

CALL CLEARSCREEN($GCLEARSCREEN)
```

See Also

- C to D
- GETBKCOLORRGB
- SETBKCOLORRGB
- SETTEXTWINDOW
- SETVIEWPORT

Building Applications: Real Coordinates Sample Program

CLEARSTATUSFPQQ

Portability Subroutine: *Clears the exception flags in the floating-point processor status word.*

Module

USE IFPORT

Syntax

```
CALL CLEARSTATUSFPQQ()
```

Description

The floating-point status word indicates which floating-point exception conditions have occurred. Intel® Fortran initially clears (sets to 0) all floating-point status flags, but as exceptions occur, the status flags accumulate until the program clears the flags again. CLEARSTATUSFPQQ will clear the flags.

CLEARSTATUSFPQQ is appropriate for use in applications that poll the floating-point status register as the method for detecting a floating-point exception has occurred.

For a full description of the floating-point status word, exceptions, and error handling, see *Floating-Point Operations*: Floating-Point Environment.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
! Program to demonstrate CLEARSTATUSFPQQ.
! This program uses polling to detect that a
! floating-point exception has occurred.
! So, build this console application with the default
! floating-point exception behavior, fpe3.
! You need to specify compiler option /debug or /Od (Windows)
! or -O0 (Linux) to get the correct results
!  PROGRAM CLEARFP
  USE IFPORT
  REAL*4 A,B,C
  INTEGER*2 STS
  A = 2.0E0
  B = 0.0E0
! Poll and display initial floating point status
  CALL GETSTATUSFPQQ(STS)
  WRITE(*,'(1X,A,Z4.4)') 'Initial fp status = ',STS
! Cause a divide-by-zero exception
! Poll and display the new floating point status
  C = A/B
  CALL GETSTATUSFPQQ(STS)
  WRITE(*,'(1X,A,Z4.4)') 'After div-by-zero fp status = ',STS
! If a divide by zero error occurred, clear the floating point
! status register so future exceptions can be detected.
  IF ((STS .AND. FPSW$ZERODIVIDE) > 0) THEN
    CALL CLEARSTATUSFPQQ()
    CALL GETSTATUSFPQQ(STS)
    WRITE(*,'(1X,A,Z4.4)') 'After CLEARSTATUSFPQQ fp status = ',STS
```

```
ENDIF
END
```

This program is available in the online samples.

See Also

- C to D
- GETSTATUSFPQQ
- SETCONTROLFPQQ
- GETCONTROLFPQQ
- SIGNALQQ

CLICKMENUQQ (W*32, W*64)

QuickWin Function: *Simulates the effect of clicking or selecting a menu command. The QuickWin application responds as though the user had clicked or selected the command.*

Module

USE IFQWIN

Syntax

`result = CLICKMENUQQ (item)`

item (Input) INTEGER(4). Constant that represents the command selected from the Window menu. Must be one of the following symbolic constants (defined in `IFQWIN.F90`):

- QWIN$STATUS - Status command
- QWIN$TILE - Tile command
- QWIN$CASCADE - Cascade command
- QWIN$ARRANGE - Arrange Icons command

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- C to D
- REGISTERMOUSEEVENT
- UNREGISTERMOUSEEVENT
- WAITONMOUSEEVENT

Building Applications: Using QuickWin Overview

CLOCK

Portability Function: *Converts a system time into an 8-character ASCII string.*

Module

USE IFPORT

Syntax

```
result = CLOCK( )
```

Results

The result type is character with a length of 8. The result is the current time in the form hh:mm:ss, using a 24-hour clock.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

character(8) whatimeisit

whatimeisit = CLOCK ()

print *, 'The current time is ',whatimeisit
```

See Also

- C to D

- DATE_AND_TIME

CLOCKX

Portability Subroutine: *Returns the processor clock to the nearest microsecond.*

Module

USE IFPORT

Syntax

```
CALL CLOCKX (clock)
```

clock	(Input) REAL(8). The current time.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

CLOSE

Statement: *Disconnects a file from a unit.*

Syntax

```
CLOSE ( [UNIT=] io-unit[, {STATUS | DISPOSE | DISP} = p] [, ERR= label] [,
IOSTAT=i-var] )
```

io-unit	(Input) an external unit specifier.
p	(Input) a scalar default character expression indicating the status of the file after it is closed. It has one of the following values: • 'KEEP' or 'SAVE' - Retains the file after the unit closes. • 'DELETE' - Deletes the file after the unit closes (unless OPEN(READONLY) is in effect). • 'PRINT' - Submits the file to the line print spooler, then retains it (sequential files only). • 'PRINT/DELETE' - Submits the file to the line print spooler, then deletes it (sequential files only). • 'SUBMIT' - Forks a process to execute the file.

- 'SUBMIT/DELETE' - Forks a process to execute the file, then deletes the file after the fork is completed.

The default is 'DELETE' for QuickWin applications (W*32, W*64) and for scratch files. For all other files, the default is 'KEEP'. Scratch files are temporary and are always deleted upon normal program termination; specifying STATUS='KEEP' for scratch files causes a run-time error.
For Windows* QuickWin applications, STATUS='KEEP' causes the child window to remain on the screen even after the unit closes. The default status is 'DELETE', which removes the child window from the screen.

label — Is the label of the branch target statement that receives control if an error occurs.

i-var — (Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The CLOSE statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

The status specified in the CLOSE statement supersedes the status specified in the OPEN statement, except that a file opened as a scratch file cannot be saved, printed, or submitted, and a file opened for read-only access cannot be deleted.

If a CLOSE statement is specified for a unit that is not open, it has no effect.

You do not need to explicitly close open files. Normal program termination closes each file according to its default status. The CLOSE statement does not have to appear in the same program unit that opened the file.

Closing unit 0 automatically reconnects unit 0 to the keyboard and screen. Closing units 5 and 6 automatically reconnects those units to the keyboard or screen, respectively.Closing the asterisk (*) unit causes a compile-time error. In Windows QuickWin applications, use CLOSE with unit 0, 5, or 6 to close the default window. If all of these units have been detached from the console (through an explicit OPEN), you must close one of these units beforehand to reestablish its connection with the console. You can then close the reconnect unit to close the default window.

If a parameter of the CLOSE statement is an expression that calls a function, that function must not cause an I/O operation or the EOF intrinsic function to be executed, because the results are unpredictable.

Example

```
C    Close and discard file:

     CLOSE (7, STATUS = 'DELETE')
```

Consider the following statement:

```
  CLOSE (UNIT=J, STATUS='DELETE', ERR=99)
```

This statement closes the file connected to unit J and deletes it. If an error occurs, control is transferred to the statement labeled 99.

See Also

- C to D
- Data Transfer I/O Statements
- Branch Specifiers

CMPLX

Elemental Intrinsic Function (Specific):
Converts the argument to complex type. This function cannot be passed as an actual argument.

Syntax

```
result = CMPLX (x[,y] [,kind])
```

x	(Input) Must be of type integer, real, or complex.
y	(Input; optional) Must be of type integer or real. It must not be present if *x* is of type complex.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is complex. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default real type.

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX(REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

CMPLX(*x*, *y*, *kind*) has the complex value whose real part is REAL(*x*, *kind*) and whose imaginary part is REAL(*y*, *kind*).

The setting of compiler options specifying real size can affect this function.

Example

CMPLX (-3) has the value (-3.0, 0.0).

CMPLX (4.1, 2.3) has the value (4.1, 2.3).

The following shows another example:

```
COMPLEX z1, z2

COMPLEX(8) z3

z1 = CMPLX(3)      ! returns the value 3.0 + i 0.0

z2 = CMPLX(3,4)    ! returns the value 3.0 + i 4.0

z3 = CMPLX(3,4,8) ! returns a COMPLEX(8) value 3.0D0 + i 4.0D0
```

See Also

- C to D
- DCMPLX
- FLOAT
- INT
- IFIX
- REAL
- SNGL

COMAddObjectReference (W*32, W*64)

***COM Function**: Adds a reference to an object's interface.*

Module

USE IFCOM

Syntax

```
result = COMAddObjectReference (iunknown)
```

iunknown	An IUnKnown interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

Results

The result type is INTEGER(4). It is the object's current reference count.

See Also

- C to D

IUnknown::AddRef in the Microsoft* Platform SDK

COMCLSIDFromProgID (W*32, W*64)

***COM Subroutine**: Passes a programmatic identifier and returns the corresponding class identifier.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMCLSIDFromProgID (prog_id,clsid,status)
```

prog_id	The programmatic identifier of type CHARACTER*(*).
clsid	The class identifier corresponding to the programmatic identifier. Must be of type GUID, which is defined in the IFWINTY module.
status	The status of the operation. It can be any status returned by CLSIDFromProgID. Must be of type INTEGER(4).

See Also

- C to D

CLSIDFromProgID in the Microsoft* Platform SDK

COMCLSIDFromString (W*32, W*64)

***COM Subroutine**: Passes a class identifier string and returns the corresponding class identifier.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMCLSIDFromString (string,clsid,status)
```

string	The class identifier string of type CHARACTER*(*).
clsid	The class identifier corresponding to the identifier string. Must be of type GUID, which is defined in the IFWINTY module.
status	The status of the operation. It can be any status returned by CLSIDFromString. Must be of type INTEGER(4).

See Also

- C to D

CLSIDFromString in the Microsoft* Platform SDK

COMCreateObjectByGUID (W*32, W*64)

***COM Subroutine**: Passes a class identifier, creates an instance of an object, and returns a pointer to the object's interface.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMCreateObjectByGUID (clsid,clsctx,iid,interface,status)
```

clsid	The class identifier of the class of object to be created. Must be of type GUID, which is defined in the IFWINTY module.
clsctx	Lets you restrict the types of servers used for the object. Must be of type INTEGER(4). Must be one of the CLSCTX_* constants defined in the IFWINTY module.
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CoCreateInstance. Must be of type INTEGER(4).

See Also

- C to D

CoCreateInstance in the Microsoft* Platform SDK

Building Applications: Getting a Pointer to an Object's Interface

COMCreateObjectByProgID (W*32, W*64)

***COM Subroutine**: Passes a programmatic identifier, creates an instance of an object, and returns a pointer to the object's IDispatch interface.*

Module

USE IFCOM

Syntax

```
CALL COMCreateObjectByProgID (prog_id,idispatch,status)
```

prog_id	The programmatic identifier of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CLSIDFromProgID or CoCreateInstance. Must be of type INTEGER(4).

See Also

- C to D
- COMCLSIDFromProgID

CoCreateInstance in the OLE section of the Microsoft* Platform SDK

Building Applications: Getting a Pointer to an Object's Interface

COMGetActiveObjectByGUID (W*32, W*64)

***COM Subroutine**: Passes a class identifier and returns a pointer to the interface of a currently active object.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMGetActiveObjectByGUID (clsid,iid,interface,status)
```

clsid	The class identifier of the class of object to be found. Must be of type GUID, which is defined in the IFWINTY module.
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by GetActiveObject. Must be of type INTEGER(4).

See Also

- C to D

GetActiveObject in the Microsoft* Platform SDK

COMGetActiveObjectByProgID (W*32, W*64)

***COM Subroutine**: Passes a programmatic identifier and returns a pointer to the IDispatch interface of a currently active object.*

Module

USE IFCOM

Syntax

```
CALL COMGetActiveObjectByProgID (prog_id,idispatch,status)
```

prog_id	The programmatic identifier of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by CLSIDFromProgID or GetActiveObject. Must be of type INTEGER(4).

Example

See the example in COMInitialize.

See Also

- C to D

CLSIDFromProgID and GetActiveObject in the Microsoft* Platform SDK

COMGetFileObject (W*32, W*64)

***COM Subroutine**: Passes a file name and returns a pointer to the IDispatch interface of an automation object that can manipulate the file.*

Module

USE IFCOM

Syntax

```
CALL COMGetFileObject (filename,idispatch,status)
```

filename	The path of the file of type CHARACTER*(*).
idispatch	An output argument that returns the object's IDispatch interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by the CreateBindCtx or MkParseDisplayName routines, or the IMoniker::BindToObject method. Must be of type INTEGER(4).

See Also

- C to D

CreateBindCtx, MkParseDisplayName, and IMonker::BindToObject in the Microsoft* Platform SDK

COMInitialize (W*32, W*64)

***COM Subroutine**: Initializes the COM library.*

Module

USE IFCOM

Syntax

```
CALL COMInitialize (status)
```

status	The status of the operation. It can be any status returned by OleInitialize. Must be of type INTEGER(4).

You must use this routine to initialize the COM library before calling any other COM or AUTO routine.

Example

Consider the following:

```
 program COMExample
use ifwin
use ifcom
use ifauto
    implicit none
    ! Variables
integer(4) word_app
integer(4) status
integer(INT_PTR_KIND()) invoke_args
call COMInitialize(status)
    ! Call GetActiveObject to get a reference to a running MS WORD application
    call COMGetActiveObjectByProgID("Word.Application", word_app, status)
if (status >= 0) then
            ! Print the active document
            invoke_args = AutoAllocateInvokeArgs()
            call AutoAddArg(invoke_args, "Copies", 2)
            status = AutoInvoke(word_app, "PrintOut", invoke_args)
            call AutoDeallocateInvokeArgs(invoke_args)
    ! Release the reference
    status = COMReleaseObject(word_app)
end if
call COMUninitialize()
    end program
```

See Also

- C to D

OleInitialize in the Microsoft* Platform SDK

COMIsEqualGUID (W*32, W*64)

***COM Function**: Determines whether two globally unique identifiers (GUIDs) are the same.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
result = COMIsEqualGUID (guid1,guid2)
```

guid1	The first GUID. Must be of type GUID, which is defined in the IFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).
guid2	The second GUID, which will be compared to *guid1*. It must be the same type of GUID as *guid1*. For example, if *guid1* is a CLSID, *guid2* must also be a CLSID.

Results

The result type is LOGICAL(4). The result is .TRUE. if the two GUIDs are the same; otherwise, .FALSE.

See Also

- C to D

IsEqualGUID in the Microsoft* Platform SDK

COMMAND_ARGUMENT_COUNT

***Inquiry Intrinsic Function (Generic):** Returns the number of command arguments.*

Syntax

```
result = COMMAND_ARGUMENT_COUNT ()
```

Results

The result is a scalar of type default integer. The result value is equal to the number of command arguments available. If there are no command arguments available, the result is 0. The command name does not count as one of the command arguments.

Example

Consider the following:

```
program echo_command_line
integer i, cnt, len, status
character c*30, b*100
call get_command (b, len, status)
if (status .ne. 0) then
   write (*,*) 'get_command failed with status = ', status
   stop
end if
write (*,*) 'command line = ', b (1:len)
call get_command_argument (0, c, len, status)
if (status .ne. 0) then
   write (*,*) 'Getting command name failed with status = ', status
   stop
end if
write (*,*) 'command name = ', c (1:len)
cnt = command_argument_count ()
write (*,*) 'number of command arguments = ', cnt
do i = 1, cnt
   call get_command_argument (i, c, len, status)
   if (status .ne. 0) then
     write (*,*) 'get_command_argument failed: status = ', status, ' arg = ', i
     stop
   end if
   write (*,*) 'command arg ', i, ' = ', c (1:len)
end do
write (*,*) 'command line processed'
```

```
end
```

If the above program is invoked with the command line " `echo_command_line.exe –o 42 –a hello b`", the following is displayed:

```
command line = echo_command_line.exe –o 42 –a hello b
command name = echo_command_line.exe
number of command arguments = 5
command arg 1 = -o
command arg 2= 42
command arg 3 = -a
command arg 4 = hello
command arg 5 = b
command line processed
```

See Also

- C to D
- GETARG
- NARGS
- IARGC
- GET_COMMAND
- GET_COMMAND_ARGUMENT

COMMITQQ

Run-Time Function: *Forces the operating system to execute any pending write operations for the file associated with a specified unit to the file's physical device.*

Module

USE IFCORE

Syntax

`result = COMMITQQ (unit)`

unit (Input) INTEGER(4). A Fortran logical unit attached to a file to be flushed from cache memory to a physical device.

Results

The result type is LOGICAL(4). If an open unit number is supplied, .TRUE. is returned and uncommitted records (if any) are written. If an unopened unit number is supplied, .FALSE. is returned.

Data written to files on physical devices is often initally written into operating-system buffers and then written to the device when the operating system is ready. Data in the buffer is automatically flushed to disk when the file is closed. However, if the program or the computer crashes before the data is transferred from buffers, the data can be lost. COMMITQQ tells the operating system to write any cached data intended for a file on a physical device to that device immediately. This is called flushing the file.

COMMITQQ is most useful when you want to be certain that no loss of data occurs at a critical point in your program; for example, after a long calculation has concluded and you have written the results to a file, or after the user has entered a group of data items, or if you are on a network with more than one program sharing the same file. Flushing a file to disk provides the benefits of closing and reopening the file without the delay.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFCORE

INTEGER unit / 10 /

INTEGER len

CHARACTER(80) stuff

OPEN(unit, FILE='COMMITQQ.TST', ACCESS='Sequential')

DO WHILE (.TRUE.)

  WRITE (*, '(A, \)') 'Enter some data (Hit RETURN to &

                       exit): '

  len = GETSTRQQ (stuff)

  IF (len .EQ. 0) EXIT

  WRITE (unit, *) stuff

  IF (.NOT. COMMITQQ(unit)) WRITE (*,*) 'Failed'

END DO

CLOSE (unit)

END
```

See Also

- C to D
- PRINT
- WRITE

COMMON

Statement: *Defines one or more contiguous areas, or blocks, of physical storage (called* `common blocks`*) that can be accessed by any of the scoping units in an executable program. COMMON statements also define the order in which variables and arrays are stored in each common block, which can prevent misaligned data items.*

Syntax

Common blocks can be named or unnamed (a *`blank common`*).

```
COMMON [ /[cname]/] var-list[[,] /[cname]/ var-list]...
```

cname	(Optional) Is the name of the common block. The name can be omitted for blank common (//).
var-list	Is a list of variable names, separated by commas. The variable must not be a dummy argument, allocatable array, automatic object, function, function result, a variable with the BIND attribute,or entry to a procedure. It must not have the PARAMETER attribute. If an object of derived type is specified, it must be a sequence type or a type with the BIND attribute.

Description

A common block is a global entity, and must not have the same name as any other global entity in the program, such as a subroutine or function.

Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The list following each successive appearance of the same common block name is treated as a continuation of the list for the block associated with that name. Consider the following COMMON statements:

```
COMMON /ralph/ ed, norton, trixie

COMMON /     / fred, ethel, lucy

COMMON /ralph/ audrey, meadows

COMMON /jerry/ mortimer, tom, mickey

COMMON melvin, purvis
```

They are equivalent to these COMMON statements:

```
COMMON /ralph/ ed, norton, trixie, audrey, meadows

COMMON         fred, ethel, lucy, melvin, purvis

COMMON /jerry/ mortimer, tom, mickey
```

A variable can appear in only one common block within a scoping unit.

If an array is specified, it can be followed by an explicit-shape array specification, each bound of which must be a constant specification expression. Such an array must not have the POINTER attribute.

A pointer can only be associated with pointers of the same type and kind parameters, and rank.

An object with the TARGET attribute can only be associated with another object with the TARGET attribute and the same type and kind parameters.

A nonpointer can only be associated with another nonpointer, but association depends on their types, as follows:

Type of Variable	Type of Associated Variable
Intrinsic numeric [1] or numeric sequence [2]	Can be of any of these types
Default character or character sequence [2]	Can be of either of these types
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1] Default integer, default real, double precision real, default complex, double complex, or default logical.

[2] If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated directly in the common list.

So, variables can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
COMMON /QUANTA/ A, Y
```

When common blocks from different program units have the same name, they share the same storage area when the units are combined into an executable program.

Entities are assigned storage in common blocks on a one-for-one basis. So, the data type of entities assigned by a COMMON statement in one program unit should agree with the data type of entities placed in a common block by another program unit. For example:

Program Unit A	Program Unit B
`COMMON CENTS`	`INTEGER(2) MONEY`
. . .	`COMMON MONEY`
	. . .

When these program units are combined into an executable program, incorrect results can occur if the 2-byte integer variable MONEY is made to correspond to the lower-addressed two bytes of the real variable CENTS.

Named common blocks must be declared to have the same size in each program unit. Blank common can have different lengths in different program units.

NOTE. If a common block is initialized by a DATA statement, the module containing the initialization must declare the common block to be its maximum defined length.

This limitation does not apply if you compile all source modules together.

Example

```
PROGRAM MyProg
COMMON i, j, x, k(10)
COMMON /mycom/ a(3)
...
END
SUBROUTINE MySub
COMMON pe, mn, z, idum(10)
COMMON /mycom/ a(3)
...
END
```

In the following example, the COMMON statement in the main program puts HEAT and X in blank common, and KILO and Q in a named common block, BLK1:

Main Program	Subprogram
`COMMON HEAT,X /BLK1/KILO,Q`	`SUBROUTINE FIGURE`
`...`	`COMMON /BLK1/LIMA,R / /ALFA,BET`
	`...`
`CALL FIGURE`	

Main Program	Subprogram
. . .	RETURN
	END

The COMMON statement in the subroutine makes ALFA and BET share the same storage location as HEAT and X in blank common. It makes LIMA and R share the same storage location as KILO and Q in BLK1.

The following example shows how a COMMON statement can be used to declare arrays:

```
COMMON / MIXED / SPOTTED(100), STRIPED(50,50)
```

The following example shows a valid association between subroutines in different program units. The object lists agree in number, type, and kind of data objects:

```
SUBROUTINE unit1
REAL(8)      x(5)
INTEGER      J
CHARACTER    str*12
TYPE(member) club(50)
COMMON / blocka / x, j, str, club
...
SUBROUTINE unit2
REAL(8)      z(5)
INTEGER      m
CHARACTER    chr*12
TYPE(member) myclub(50)
COMMON / blocka / z, m, chr, myclub
...
```

See also the example for BLOCK DATA.

See Also

- C to D
- BLOCK DATA

- DATA
- MODULE
- EQUIVALENCE
- Specification expressions
- Storage association
- Interaction between COMMON and EQUIVALENCE Statements

COMPLEX Statement

Statement: *Specifies the COMPLEX data type.*

Syntax

```
COMPLEX

COMPLEX([KIND=] n)

COMPLEX*s

DOUBLE COMPLEX
```

n	Is kind 4, 8, or 16.
s	Is 8, 16, or 32. COMPLEX(4) is specified as COMPLEX*8; COMPLEX(8) is specified as COMPLEX*16; COMPLEX(16) is specified as COMPLEX*32.

If a kind parameter is specified, the complex constant has the kind specified. If no kind parameter is specified, the kind of both parts is default real, and the constant is of type default complex.

DOUBLE COMPLEX is COMPLEX(8). No kind parameter is permitted for data declared with type DOUBLE COMPLEX.

Example

```
COMPLEX ch

COMPLEX (KIND=4),PRIVATE :: zz, yy   !equivalent to COMPLEX*8 zz, yy

COMPLEX(8) ax, by        !equivalent to COMPLEX*16 ax, by

COMPLEX (kind(4)) y(10)

complex (kind=8)  x, z(10)
```

See Also

- C to D

- DOUBLE COMPLEX
- Complex Data Type
- COMPLEX(4) Constants
- COMPLEX(8) or DOUBLE COMPLEX Constants
- Data Types, Constants, and Variables

COMPLINT, COMPLREAL, COMPLLOG

Portability Functions: *Return a BIT-WISE complement or logical .NOT. of the argument.*

Module

USE IFPORT

Syntax

```
result = COMPLINT (intval)
result = COMPREAL (realval)
result = COMPLLOG (logval)
```

intval	(Input) INTEGER(4).
realval	(Input) REAL(4).
logval	(Input) LOGICAL(4).

Results

The result is INTEGER(4) for COMPLINT, REAL(4) for COMPLREAL and LOGICAL(4) for COMPLLOG with a value that is the bitwise complement of the argument.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

COMQueryInterface (W*32, W*64)

COM Subroutine*: Passes an interface identifier and returns a pointer to an object's interface.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMQueryInterface (iunknown,iid,interface,status)
```

iunknown	An IUnknown interface pointer. Must be of type INTEGER(4).
iid	The interface identifier of the interface being requested. Must be of type GUID, which is defined in the IFWINTY module.
interface	An output argument that returns the object's interface pointer. Must be of type INTEGER(INT_PTR_KIND()).
status	The status of the operation. It can be any status returned by the IUnknown method QueryInterface. Must be of type INTEGER(4).

See Also

- C to D

IUnknown::QueryInterface in the Microsoft* Platform SDK

Building Applications: Getting a Pointer to an Object's Interface

COMReleaseObject (W*32, W*64)

***COM Function**: Indicates that the program is done with a reference to an object's interface.*

Module

USE IFCOM

Syntax

```
result = COMReleaseObject (iunknown)
```

iunknown	An IUnknown interface pointer. Must be of type INTEGER(INT_PTR_KIND()).

Results

The result type is INTEGER(4). It is the object's current reference count.

Example

See the example in COMInitialize.

COMStringFromGUID (W*32, W*64)

***COM Subroutine**: Passes a globally unique identifier (GUID) and returns a string of printable characters.*

Module

USE IFCOM

USE IFWINTY

Syntax

```
CALL COMStringFromGUID (guid,string,status)
```

guid	The GUID to be converted. Must be of type GUID, which is defined in the IFWINTY module. It can be any type of GUID, including a class identifier (CLSID), or an interface identifier (IID).
string	A character variable of type CHARACTER*(*) that receives the string representation of the GUID. The length of the character variable should be at least 38.
status	The status of the operation. If the string is too small to contain the string representation of the GUID, the value is zero. Otherwise, the value is the number of characters in the string representation of the GUID. Must be of type INTEGER(4).

The string representation of a GUID has a format like that of the following:

[c200e360-38c5-11ce-ae62-08002b2b79ef]

where the successive fields break the GUID into the form DWORD-WORD-WORD-WORD-WORD.DWORD covering the 128-bit GUID. The string includes enclosing braces, which are an OLE convention.

See Also

- C to D

StringFromGUID2 in the Microsoft* Platform SDK

COMUninitialize (W*32, W*64)

***COM Subroutine**: Uninitializes the COM library.*

Module

USE IFCOM

Syntax

```
CALL COMUninitialize( )
```

When using COM routines, this must be the last routine called.

Example

See the example in COMInitialize.

CONJG

Elemental Intrinsic Function (Generic):
Calculates the conjugate of a complex number.

Syntax

```
result = CONJG (z)
```

z (Input) Must be of type complex.

Results

The result type is the same as *z*. If *z* has the value (x, y), the result has the value (x, -y).

Specific Name	Argument Type	Result Type
CONJG	COMPLEX(4)	COMPLEX(4)
DCONJG	COMPLEX(8)	COMPLEX(8)
QCONJG	COMPLEX(16)	COMPLEX(16)

Example

CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

CONJG ((1.0, -4.2)) has the value (1.0, 4.2).

The following shows another example:

```
COMPLEX z1

COMPLEX(8) z2

z1 = CONJG((3.0, 5.6))       ! returns (3.0, -5.6)

z2 = DCONJG((3.0D0, 5.6D0))  ! returns (3.0D0, -5.6D0)
```

See Also

- C to D
- AIMAG

CONTAINS

Statement: *Separates the body of a main program, module, or external subprogram from any internal or module procedures it may contain. It is not executable.*

Syntax

```
CONTAINS
```

Any number of internal procedures can follow a CONTAINS statement, but a CONTAINS statement cannot appear in the internal procedures themselves.

Example

```
PROGRAM OUTER

   REAL, DIMENSION(10) :: A

   . . .

   CALL INNER (A)

CONTAINS

   SUBROUTINE INNER (B)

   REAL, DIMENSION(10) :: B

   . . .

   END SUBROUTINE INNER

END PROGRAM OUTER
```

See Also

- C to D
- Internal Procedures
- Modules and Module Procedures
- Main Program

CONTINUE

Statement: *Primarily used to terminate a labeled DO construct when the construct would otherwise end improperly with either a GO TO, arithmetic IF, or other prohibited control statement.*

Syntax

```
CONTINUE
```

The statement by itself does nothing and has no effect on program results or execution sequence.

Example

The following example shows a CONTINUE statement:

```
    DO 150 I = 1,40
40  Y = Y + 1
    Z = COS(Y)
    PRINT *, Z
    IF (Y .LT. 30) GO TO 150
    GO TO 40
150 CONTINUE
```

The following shows another example:

```
     DIMENSION narray(10)
     DO 100 n = 1, 10
     narray(n) = 120
100  CONTINUE
```

See Also

- C to D

- END DO
- DO
- Execution Control

COPYIN

Parallel Directive Clause: *Specifies that the data in the master thread of the team is to be copied to the thread private copies of the common block at the beginning of the parallel region.*

Syntax

```
COPYIN (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The COPYIN clause applies only to common blocks declared as THREADPRIVATE.

You do not need to specify the whole THREADPRIVATE common block, you can specify named variables within the common block.

COPYPRIVATE

Parallel Directive Clause: *Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. The COPYPRIVATE clause can only appear in the END SINGLE directive.*

Syntax

```
COPYPRIVATE (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables in the list must not appear in a PRIVATE or FIRSTPRIVATE clause for the SINGLE directive construct.

If the directive is encountered in the dynamic extent of a parallel region, variables in the list must be private in the enclosing context.

If a common block is specified, it must be declared as THREADPRIVATE; the effect is the same as if the variable names in its common block object list were specified.

The effect of the COPYPRIVATE clause on the variables in its list occurs after the execution of the code enclosed within the SINGLE construct, and before any threads in the team have left the barrier at the end of the construct.

COS

Elemental Intrinsic Function (Generic):
Produces the cosine of x.

Syntax

```
result = COS (x)
```

x	(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi. If *x* is of type complex, its real part is regarded as a value in radians.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COS	REAL(4)	REAL(4)
DCOS	REAL(8)	REAL(8)
QCOS	REAL(16)	REAL(16)
CCOS [1]	COMPLEX(4)	COMPLEX(4)
CDCOS[2]	COMPLEX(8)	COMPLEX(8)
CQCOS	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CCOS.

[2]This function can also be specified as ZCOS.

Example

COS (2.0) has the value -0.4161468.

COS (0.567745) has the value 0.8431157.

See Also

- C to D

Optimizing Applications: Types of Vectorized Loops

COSD

Elemental Intrinsic Function (Generic):
Produces the cosine of x.

Syntax

```
result = COSD (x)
```

x	(Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COSD	REAL(4)	REAL(4)
DCOSD	REAL(8)	REAL(8)
QCOSD	REAL(16)	REAL(16)

Example

COSD (2.0) has the value 0.9993908.

COSD (30.4) has the value 0.8625137.

COSH

Elemental Intrinsic Function (Generic):
Produces a hyperbolic cosine.

Syntax

```
result = COSH (x)
```

x (Input) Must be of type real.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COSH	REAL(4)	REAL(4)
DCOSH	REAL(8)	REAL(8)
QCOSH	REAL(16)	REAL(16)

Example

COSH (2.0) has the value 3.762196.

COSH (0.65893) has the value 1.225064.

COTAN

Elemental Intrinsic Function (Generic):
Produces the cotangent of x.

Syntax

```
result = COTAN (x)
```

x (Input) Must be of type real; it cannot be zero. It must be in radians and is treated as modulo 2*pi.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COTAN	REAL(4)	REAL(4)
DCOTAN	REAL(8)	REAL(8)
QCOTAN	REAL(16)	REAL(16)

Example

COTAN (2.0) has the value -4.576575E-01.

COTAN (0.6) has the value 1.461696.

COTAND

Elemental Intrinsic Function (Generic):
Produces the cotangent of x.

Syntax

```
result = COTAND (x)
```

x (Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
COTAND	REAL(4)	REAL(4)
DCOTAND	REAL(8)	REAL(8)
QCOTAND	REAL(16)	REAL(16)

Example

COTAND (2.0) has the value 0.2863625E+02.

COTAND (0.6) has the value 0.9548947E+02.

COUNT

Transformational Intrinsic Function (Generic):
Counts the number of true elements in an entire array or in a specified dimension of an array.

Syntax

```
result = COUNT (mask[,dim][, kind])
```

mask	(Input) Must be a logical array.
dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *mask*.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is an array or a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result is a scalar if *dim* is omitted or *mask* has rank one. A scalar result has a value equal to the number of true elements of *mask*. If *mask* has size zero, the result is zero.

An array result has a rank that is one less than *mask*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2,..., d_n) is the shape of *mask*.

Each element in an array result equals the number of elements that are true in the one dimensional array defined by *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n).

Example

COUNT ((/.TRUE., .FALSE., .TRUE./)) has the value 2 because two elements are true.

COUNT ((/.TRUE., .TRUE., .TRUE./)) has the value 3 because three elements are true.

A is the array

```
[ 1  5  7 ]
[ 3  6  8 ]
```

and B is the array

```
[ 0  5  7 ]
[ 2  6  9 ].
```

COUNT (A .NE. B, DIM=1) tests to see how many elements in each column of A are not equal to the elements in the corresponding column of B. The result has the value (2, 0, 1) because:

- The first column of A and B have 2 elements that are not equal.
- The second column of A and B have 0 elements that are not equal.
- The third column of A and B have 1 element that is not equal.

COUNT (A .NE. B, DIM=2) tests to see how many elements in each row of A are not equal to the elements in the corresponding row of B. The result has the value (1, 2) because:

- The first row of A and B have 1 element that is not equal.
- The second row of A and B have 2 elements that are not equal.

The following shows another example:

```
LOGICAL mask (2, 3)

INTEGER AR1(3), AR2(2), I

mask = RESHAPE((/.TRUE., .TRUE., .FALSE., .TRUE., &

                .FALSE., .FALSE./),(/2,3/))

!

! mask is the array   true false false

!                     true true false

!AR1 = COUNT(mask,DIM=1) ! counts true elements by

                         ! column yielding [2 1 0]

AR2 = COUNT(mask,DIM=2) ! counts true elements by row

                         ! yielding [1 2]

I = COUNT( mask)         ! returns 3
```

See Also

- C to D
- ALL
- ANY

CPU_TIME

Intrinsic Subroutine (Generic): *Returns a processor-dependent approximation of the processor time in seconds. This is a new intrinsic procedure in Fortran 95. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL CPU_TIME (time)
```

time	(Output) Must be scalar and of type real.

If a meaningful time cannot be returned, a processor-dependent negative value is returned.

Example

Consider the following:

```
REAL time_begin, time_end
...
CALL CPU_TIME ( time_begin )
!
!task to be timed
!
CALL CPU_TIME ( time_end )
PRINT (*,*) 'Time of operation was ', time_end - time_begin, ' seconds'
```

CRITICAL

OpenMP* Fortran Compiler Directive: *Restricts access to a block of code to only one thread at a time.*

Syntax

```
c$OMP CRITICAL [(name)]
   block
c$OMP END CRITICAL [(name)]
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
name	Is the name of the critical section.
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section having the same name. All unnamed CRITICAL directives map to the same name.

If a name is specified in the CRITICAL directive, the same name must appear in the corresponding END CRITICAL directive. If no name appears in the CRITICAL directive, no name can appear in the corresponding END CRITICAL directive.

Critical section names are global entities of the program. If the name specified conflicts with any other entity, the behavior of the program is undefined.

Example

The following example shows a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation is placed in a critical section.

Because there are two independent queues in this example, each queue is protected by CRITICAL directives having different names, XAXIS and YAXIS, respectively:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
c$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
c$OMP END   CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
c$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
c$OMP END   CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
c$OMP END PARALLEL
```

See Also

- C to D

- OpenMP Fortran Compiler Directives

CSHIFT

Transformational Intrinsic Function (Generic):
Performs a `circular` shift on a rank-one array, or performs circular shifts on all the complete rank-one sections (vectors) along a given dimension of an array of rank two or greater.

Syntax

Elements shifted off one end are inserted at the other end. Different sections can be shifted by different amounts and in different directions.

`result = CSHIFT (array,shift [,dim])`

array	(Input) Array whose elements are to be shifted. It can be of any data type.
shift	(Input) The number of positions shifted. Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
dim	(Input; optional) Optional dimension along which to perform the shift. Must be a scalar integer with a value in the range 1 to n, where n is the rank of array. If *dim* is omitted, it is assumed to be 1.

Results

The result is an array with the same type and kind parameters, and shape as *array*.

If *array* has rank one, element i of the result is *array*(1 + MODULO (i + *shift*- 1, SIZE (*array*))). (The same shift is applied to each element.)

If *array* has rank greater than one, each section (s_1,s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2,..., s_{dim-1}, s_{dim+1},..., s_n), if *shift* is an array

The value of *shift* determines the amount and direction of the circular shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns). A zero *shift* value causes no shift.

Example

V is the array (1, 2, 3, 4, 5, 6).

CSHIFT (V, SHIFT=2) shifts the elements in V circularly to the *left* by 2 positions, producing the value (3, 4, 5, 6, 1, 2). 1 and 2 are shifted off the beginning and inserted at the end.

CSHIFT (V, SHIFT= -2) shifts the elements in V circularly to the *right* by 2 positions, producing the value (5, 6, 1, 2, 3, 4). 5 and 6 are shifted off the end and inserted at the beginning.

M is the array

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ].
```

CSHIFT (M, SHIFT = 1, DIM = 2) produces the result

```
[ 2  3  1 ]
[ 5  6  4 ]
[ 8  9  7 ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 2 positions. The elements shifted off the beginning are inserted at the end.

CSHIFT (M, SHIFT = -1, DIM = 1) produces the result

```
[ 7  8  9 ]
[ 1  2  3 ]
[ 4  5  6 ].
```

Each element in columns 1, 2, and 3 is shifted down by 1 position. The elements shifted off the end are inserted at the beginning.

CSHIFT (M, SHIFT = (/1, -1, 0/), DIM = 2) produces the result

```
[ 2  3  1 ]
[ 6  4  5 ]
[ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 position; each element in row 2 is shifted to the *right* by 1 position; no element in row 3 is shifted at all.

The following shows another example:

```
INTEGER array (3, 3), AR1(3, 3), AR2 (3, 3)
DATA array /1, 4, 7, 2, 5, 8, 3, 6, 9/
!
! array is   1 2 3
!            4 5 6
!            7 8 9
!AR1 = CSHIFT(array, 1, DIM = 1) ! shifts all columns
                                  ! by 1 yielding
                                  !         4 5 6
                                  !         7 8 9
                                  !         1 2 3
                                  !
AR2=CSHIFT(array,shift=(/-1, 1, 0/),DIM=2) ! shifts
                                  ! each row separately
                                  ! by the amount in
                                  ! shift yielding
                                  !         3 1 2
                                  !         5 6 4
                                  !         7 8 9
```

See Also

- C to D
- EOSHIFT
- ISHFT
- ISHFTC

CSMG

Portability Function: *Performs an effective BIT-WISE store under mask.*

Module

USE IFPORT

Syntax

`result = CSMG (x,y,z)`

x, *y*, *z*	(Input) INTEGER(4).

Results

The result type is INTEGER(4). The result is equal to the following expression:

```
(x & z) | (y & ~z )
```

where "&" is a bitwise AND operation, | - bitwise OR, ~ - bitwise NOT.

The function returns the value based on the following rule: when a bit in *z* is 1, the output bit is taken from *x*. When a bit in *z* is zero, the corresponding output bit is taken from *y*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

CTIME

Portability Function: *Converts a system time into a 24-character ASCII string.*

Module

USE IFPORT

Syntax

`result = CTIME (stime)`

stime	(Input) INTEGER(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

Results

The result is a value in the form Mon Jan 31 04:37:23 1994. Hours are expressed using a 24-hour clock.

The value of *stime* can be determined by calling the TIME function. CTIME(TIME( )) returns the current time and date.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

character (24) systime

systime = CTIME (TIME( ))

print *, 'Current date and time is ',systime
```

See Also

- C to D
- DATE_AND_TIME

CYCLE

Statement: *Interrupts the current execution cycle of the innermost (or named) DO construct.*

Syntax

CYCLE [*name*]

name	(Optional) Is the name of the DO construct.

Description

When a CYCLE statement is executed, the following occurs:

1. The current execution cycle of the named (or innermost) DO construct is terminated.

   If a DO construct name is specified, the CYCLE statement must be within the range of that construct.

2. The iteration count (if any) is decremented by 1.

3. The DO variable (if any) is incremented by the value of the increment parameter (if any).

4. A new iteration cycle of the DO construct begins.

Any executable statements following the CYCLE statement (including a labeled terminal statement) are not executed.

A CYCLE statement can be labeled, but it cannot be used to terminate a DO construct.

Example

The following example shows a CYCLE statement:

```
DO I =1, 10
  A(I) = C + D(I)
  IF (D(I) < 0) CYCLE    ! If true, the next statement is omitted
  A(I) = 0               ! from the loop and the loop is tested again.
END DO
```

The following shows another example:

```
      sample_loop: do i = 1, 5
           print *,i
             if( i .gt. 3 ) cycle sample_loop
           print *,i
           end do sample_loop
           print *,'done!'
!output:
!     1
!     1
!     2
!     2
!     3
!     3
!     4
!     5
!     done!
```

See Also

- C to D
- DO
- DO WHILE
- DO Constructs

DATA

Statement: *Assigns initial values to variables before program execution.*

Syntax

DATA *var-list /clist/* [[,] *var-list /clist/*]...

var-list Is a list of variables or implied-DO lists, separated by commas.

	Subscript expressions and expressions in substring references must be initialization expressions. An implied-DO list in a DATA statement takes the following form: (*do-list*, *var*= *expr1*, *expr2*[, *expr3*])
do-list	Is a list of one or more array elements, substrings, scalar structure components, or implied-DO lists, separated by commas. Any array elements or scalar structure components must not have a constant parent.
var	Is the name of a scalar integer variable (the implied-DO variable).
expr	Is a list of variables or implied-DO lists, separated by commas.
clist	Is a list of variables or implied-DO lists, separated by commas. A constant can be specified in the form r*constant, where *r* is a repeat specification. It is a nonnegative scalar integer constant (with no kind parameter). If it is a named constant, it must have been declared previously in the scoping unit or made accessible through use or host association. If *r* is omitted, it is assumed to be 1.

Description

A variable can be initialized only once in an executable program. A variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration which may change the implicit typing.

The number of constants in *c-list* must equal the number of variables in *var-list*. The constants are assigned to the variables in the order in which they appear (from left to right).

The following objects cannot be initialized in a DATA statement:

- A dummy argument
- A function
- A function result
- An automatic object
- An allocatable array
- A variable that is accessible by use or host association
- A variable in a named common block (unless the DATA statement is in a block data program unit)
- A variable in blank common

Except for variables in named COMMON blocks, a named variable has the SAVE attribute if any part of it is initialized in a DATA statement. You can confirm this property by specifying the variable in a SAVE statement or a type declaration statement containing the SAVE attribute.

When an unsubscripted array name appears in a DATA statement, values are assigned to every element of that array in the order of subscript progression. If the associated constant list does not contain enough values to fill the array, a warning is issued and the remaining array elements become undefined.

Array element values can be initialized in three ways: by name, by element, or by an implied-DO list (interpreted in the same way as a DO construct).

The following conversion rules and restrictions apply to variable and constant list items:

- If the constant and the variable are both of numeric type, the following conversion occurs:
  - The constant value is converted to the data type of the variable being initialized, if necessary.
  - When a binary, octal, or hexadecimal constant is assigned to a variable or array element, the number of digits that can be assigned depends on the data type of the data item. If the constant contains fewer digits than the capacity of the variable or array element, the constant is extended on the left with zeros. If the constant contains more digits than can be stored, the constant is truncated on the left. An error results if any nonzero digits are truncated.
- If the constant and the variable are both of character type, the following conversion occurs:
  - If the length of the constant is less than the length of the variable, the rightmost character positions of the variable are initialized with blank characters.
  - If the length of the constant is greater than the length of the variable, the character constant is truncated on the right.
- If the constant is of numeric type and the variable is of character type, the following restrictions apply:
  - The character variable must have a length of one character.
  - The constant must be an integer, binary, octal, or hexadecimal constant, and must have a value in the range 0 through 255.

  When the constant and variable conform to these restrictions, the variable is initialized with the character that has the ASCII code specified by the constant. (This lets you initialize a character object to any 8-bit ASCII code.)

- If the constant is a Hollerith or character constant, and the variable is a numeric variable or numeric array element, the number of characters that can be assigned depends on the data type of the data item.

  If the Hollerith or character constant contains fewer characters than the capacity of the variable or array element, the constant is extended on the right with blank characters. If the constant contains more characters than can be stored, the constant is truncated on the right.

Example

The following example shows the three ways that DATA statements can initialize array element values:

```
DIMENSION A(10,10)

DATA A/100*1.0/    ! initialization by name

DATA A(1,1), A(10,1), A(3,3) /2*2.5, 2.0/ ! initialization by element

DATA ((A(I,J), I=1,5,2), J=1,5) /15*1.0/  ! initialization by implied-DO list
```

The following example shows DATA statements containing structure components:

```
TYPE EMPLOYEE

  INTEGER ID

  CHARACTER(LEN=40) NAME

END TYPE EMPLOYEE

TYPE(EMPLOYEE) MAN_NAME, CON_NAME

DATA MAN_NAME / EMPLOYEE(417, 'Henry Adams') /

DATA CON_NAME%ID, CON_NAME%NAME /891, "David James"/
```

In the following example, the first DATA statement assigns zero to all 10 elements of array A, and four asterisks followed by two blanks to the character variable STARS:

```
INTEGER A(10), B(10)

CHARACTER BELL, TAB, LF, FF, STARS*6

DATA A,STARS /10*0,'****'/

DATA BELL,TAB,LF,FF /7,9,10,12/

DATA (B(I), I=1,10,2) /5*1/
```

In this case, the second DATA statement assigns ASCII control character codes to the character variables BELL, TAB, LF, and FF. The last DATA statement uses an implied-DO list to assign the value 1 to the odd-numbered elements in the array B.

The following shows another example:

```
    INTEGER n, order, alpha, list(100)
    REAL coef(4), eps(2),
    pi(5), x(5,5)
    CHARACTER*12 help
    COMPLEX*8 cstuff
    DATA  n /0/, order /3/
    DATA  alpha /'A'/
    DATA  coef /1.0, 2*3.0, 1.0/, eps(1) /.00001/
    DATA  cstuff /(-1.0, -1.0)/
!    The following example initializes diagonal and below in
!    a 5x5 matrix:
    DATA  ((x(j,i), i=1,j), j=1,5) / 15*1.0 /
    DATA  pi / 5*3.14159 /
    DATA  list / 100*0 /
    DATA  help(1:4), help(5:8), help(9:12) /3*'HELP'/
```

Consider the following:

```
 CHARACTER (LEN = 10) NAME
 INTEGER, DIMENSION (0:9) :: MILES
 REAL, DIMENSION (100, 100) :: SKEW
 TYPE (MEMBER) MYNAME, YOURS
 DATA NAME / 'JOHN DOE' /, miles / 10*0 /
 DATA ((SKEW (k, j), j = 1, k), k = 1, 100) / 5050*0.0 /
 DATA ((SKEW (k, j), j = k + 1, 100), k = 1, 99) / 4950*1.0 /
 DATA MYNAME / MEMBER (21, 'JOHN SMITH') /
 DATA YOURS % age, YOURS % name / 35, 'FRED BROWN' /
```

In this example, the character variable NAME is initialized with the value JOHN DOE with two trailing blanks to fill out the declared length of the variable. The ten elements of MILES are initialized to zero. The two-dimensional array SKEW is initialized so that its lower triangle is zero and its upper triangle is one. The structures MYNAME and YOURS are declared using the derived type MEMBER. The derived-type variable MYNAME is initialized by a structure constructor. The derived-type variable YOURS is initialized by supplying a separate value for each component.

The first DATA statement in the previous example could also be written as:

```
DATA name / 'JOHN DOE' /

DATA miles / 10*0 /
```

As a Fortran 95 feature, a pointer can be initialized as disassociated by using a DATA statement. For example:

```
INTEGER, POINTER :: P

DATA P/NULL( )/

END
```

See Also

- C to D
- CHARACTER
- INTEGER
- REAL
- COMPLEX
- COMMON
- Data Types, Constants, and Variables
- I/O Lists
- Derived Data Types

Building Applications: Allocating Common Blocks

DATE Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns the current date as set within the system. DATE can be used as an intrinsic subroutine or as a portability*

function or subroutine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.

Syntax

`CALL DATE (buf)`

buf	(Output) Is a variable, array, or arrayelement of any data type, or a character substring. It must contain at least nine bytes ofstorage.

The date is returned as a 9-byte ASCII character string taking the form dd-mmm-yy, where:

dd	is the 2-digit date
mmm	is the 3-letter month
yy	is the last two digits of the year

If *buf* is of numeric type and smaller than 9 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 9 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

CAUTION. The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

```
CHARACTER*1 DAY(9)
…
CALL DATE (DAY)
```

The length of the first array element in CHARACTER array DAY is passed to the DATE subroutine. The subroutine then truncates the date to fit into the 1-character element, producing an incorrect result.

See Also

- C to D
- DATE_AND_TIME

- DATE portability routine

DATE Portability Routine

Portability Function or Subroutine: *Returns the current system date. DATE can be used as an intrinsic subroutine or as a portability function or subroutine. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

Function Syntax:

```
result = DATE( )
```

Subroutine Syntax:

```
CALL DATE (dstring)
```

dstring	(Output) CHARACTER. Is a variable or array containing at least nine bytes of storage.

DATE in its function form returns a CHARACTER string of length 8 in the form mm/dd/yy, where mm, dd, and yy are two-digit representations of the month, day, and year, respectively.

DATE in its subroutine form returns *dstring* in the form dd-mmm-yy, where dd is a two-digit representation of the current day of the month, mmm is a three-character abbreviation for the current month (for example, Jan) and yy are the last two digits of the current year.

CAUTION. The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
!If today's date is March 02, 2000, the following
!code prints "02-Mar-00"
CHARACTER(9) TODAY
CALL DATE(TODAY)
PRINT *, TODAY
!The next line prints "03/02/00"
PRINT *, DATE( )
```

See Also

- C to D
- DATE_AND_TIME
- DATE intrinsic procedure

DATE4

Portability Subroutine: *Returns the current system date.*

Module

USE IFPORT

Syntax

CALL DATE4 (*datestr*)

datestr (Output) CHARACTER.

This subroutine returns *datestr* in the form dd-mmm-yyyy, where dd is a two-digit representation of the current day of the month, mmm is a three-character abbreviation for the current month (for example, Jan) and yyyy are the four digits of the current year.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

DATE_AND_TIME

Intrinsic Subroutine (Generic): *Returns character data on the real-time clock and date in a form compatible with the representations defined in Standard ISO 8601:1988. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL DATE_AND_TIME ( [date] [,time] [,zone] [,values] )
```

date (Output; optional) Must be scalar and of type default character; its length must be at least 8 to contain the complete value. Its leftmost 8 characters are set to a value of the form CCYYMMDD, where:

CC	Is the century
YY	Is the year within the century
MM	Is the month within the year
DD	Is the day within the month

time (Output; optional) Must be scalar and of type default character; its length must be at least 10 to contain the complete value. Its leftmost 10 characters are set to a value of the form hhmmss.sss, where:

hh	Is the hour of the day
mm	Is the minutes of the hour
ss.sss	Is the seconds and milliseconds of the minute

zone (Output; optional) Must be scalar and of type default character; its length must be at least 5 to contain the complete value. Its leftmost 5 characters are set to a value of the form hhmm, where

hh and *mm* are the time difference with respect to Coordinated Universal Time (UTC) in hours and parts of an hour expressed in minutes, respectively.
UTC (also known as Greenwich Mean Time) is defined by CCIR Recommendation 460-2.

values (Output; optional) Must be of type integer. One-dimensional array with size of at least 8. The values returned in *values* are as follows:

values(1)	Is the 4-digit year
values(2)	Is the month of the year
values(3)	Is the day of the month
values(4)	Is the time difference with respect to Coordinated Universal Time (UTC) in minutes
values(5)	Is the hour of the day (range 0 to 23) - local time
values(6)	Is the minutes of the hour (range 0 to 59) - local time
values(7)	Is the seconds of the minute (range 0 to 59) - local time
values(8)	Is the milliseconds of the second (range 0 to 999) - local time

Example

Consider the following example executed on 2000 March 28 at 11:04:14.5:

```
INTEGER DATE_TIME (8)

CHARACTER (LEN = 12) REAL_CLOCK (3)

CALL DATE_AND_TIME (REAL_CLOCK (1), REAL_CLOCK (2), &
                    REAL_CLOCK (3), DATE_TIME)
```

This assigns the value "20000328" to REAL_CLOCK (1), the value "110414.500" to REAL_CLOCK (2), and the value "-0500" to REAL_CLOCK (3). The following values are assigned to DATE_TIME: 2000, 3, 28, -300, 11, 4, 14, and 500.

The following shows another example:

```
CHARACTER(10) t

CHARACTER(5) z

CALL DATE_AND_TIME(TIME = t, ZONE = z)
```

See Also

- C to D
- GETDAT
- GETTIM
- IDATE intrinsic procedure
- FDATE
- TIME intrinsic procedure
- ITIME
- RTC
- CLOCK

DBESJ0, DBESJ1, DBESJN, DBESY0, DBESY1, DBESYN

Portability Functions: *Compute the double-precision values of Bessel functions of the first and second kinds.*

Module

USE IFPORT

Syntax

```
result = DBESJ0 (value)
result = DBESJ1 (value)
result = DBESJN (n, value)
result = DBESY0 (posvalue)
result = DBESY1 (posvalue)
result = DBESYN (n, posvalue)
```

value	(Input) REAL(8). Independent variable for a Bessel function.
n	(Input) Integer. Specifies the order of the selected Bessel function computation.
posvalue	(Input) REAL(8). Independent variable for a Bessel function. Must be greater than or equal to zero.

Results

DBESJ0, DBESJ1, and DBESJN return Bessel functions of the first kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

DBESY0, DBESY1, and DBESYN return Bessel functions of the second kind, orders 0, 1, and *n*, respectively, with the independent variable *posvalue*.

Negative arguments cause DBESY0, DBESY1, and DBESYN to return a huge negative value.

Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions*(Abramowitz and Stegun. Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
    USE IFPORT
    real(8) besnum, besout
10  read *, besnum
    besout = dbesj0(besnum)
    print *, 'result is ',besout
    goto 10
    end
```

See Also

- C to D
- BESJ0, BESJ1, BESJN, BESY0, BESY1, BESYN

DBLE

Elemental Intrinsic Function (Generic):
Converts a number to double-precision real type.

Syntax

```
result = DBLE (a)
```

a	(Input) Must be of type integer, real, or complex.

Results

The result type is double precision real (by default, REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type double precision, the result is the value of the *a* with no conversion (DBLE(*a*) = *a*).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a double precision value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a double precision value can contain.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(8)
	INTEGER(2)	REAL(8)
	INTEGER(4)	REAL(8)
	INTEGER(8)	REAL(8)
DBLE[2]	REAL(4)	REAL(8)
	REAL(8)	REAL(8)
DBLEQ	REAL(16)	REAL(8)
	COMPLEX(4)	REAL(8)
	COMPLEX(8)	REAL(8)
	COMPLEX(16)	REAL(8)

[1]These specific functions cannot be passed as actual arguments.

[2] The setting of compiler options specifying double size can affect DBLE.

Example

DBLE (4) has the value 4.0.

DBLE ((3.4, 2.0)) has the value 3.4.

See Also

- C to D
- FLOAT
- SNGL
- REAL
- CMPLX

DCLOCK

Portability Function: *Returns the elapsed time in seconds since the start of the current process.*

Module

USE IFPORT

Syntax

```
result = DCLOCK( )
```

Results

The result type is REAL(8). This routine provides accurate timing to the nearest microsecond, taking into account the frequency of the processor where the current process is running. You can obtain equivalent results using standard Fortran by using the CPU_TIME intrinsic subroutine.

Note that the first call to DCLOCK performs calibration.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

DOUBLE PRECISION START_TIME, STOP_TIME, DCLOCK

EXTERNAL DCLOCK

START_CLOCK = DCLOCK()

CALL FOO()

STOP_CLOCK = DCLOCK()

PRINT *, 'foo took:', STOP_CLOCK - START_CLOCK, 'seconds.'
```

See Also

- C to D
- DATE_AND_TIME
- CPU_TIME

DCMPLX

Elemental Intrinsic Function (Specific):
Converts the argument to double complex type. This function cannot be passed as an actual argument.

Syntax

```
result = DCMPLX (x[,y])
```

x	(Input) Must be of type integer, real, or complex.
y	(Input; optional) Must be of type integer or real. It must not be present if *x* is of type complex.

Results

The result type is double complex (COMPLEX(8) or COMPLEX*16).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX(REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

DCMPLX(*x*, *y*) has the complex value whose real part is REAL(*x*, KIND=8) and whose imaginary part is REAL(*y*, KIND=8).

Example

DCMPLX (-3) has the value (-3.0, 0.0).

DCMPLX (4.1, 2.3) has the value (4.1, 2.3).

See Also

- C to D
- CMPLX
- FLOAT
- INT
- IFIX
- REAL

- SNGL

DEALLOCATE

Statement: *Frees the storage allocated for allocatable arrays and pointer targets (and causes the pointers to become disassociated).*

Syntax

```
DEALLOCATE (object[,object]...[, alloc-opt])
```

object	Is a structure component or the name of a variable, and must be a pointer or allocatable array.	
alloc-opt	(Output) Is one of the following:	
	STAT=*sv*	*sv* is a scalar integer variable in which the status of the deallocation is stored.
	ERRMSG=*ev*	*ev* is a scalar default character value in which an error condition is stored if such a condition occurs.

Description

If a STAT variableor ERRMSG variable is specified, it must not be deallocated in the DEALLOCATE statement in which it appears. If the deallocation is successful, the variable is set to zero. If the deallocation is not successful, an error condition occurs, and the variable is set to a positive integer value (representing the run-time error); the ERRMSG variable contains the error condition. If no STAT variable is specified and an error condition occurs, program execution terminates.

It is recommended that all explicitly allocated storage be explicitly deallocated when it is no longer needed.

To disassociate a pointer that was not associated with the ALLOCATE statement, use the NULLIFY statement.

For a list of run-time errors, see *Building Applications*.

Example

The following example shows deallocation of an allocatable array:

```
INTEGER ALLOC_ERR

REAL, ALLOCATABLE :: A(:), B(:,:)

...

ALLOCATE (A(10), B(-2:8,1:5))

...

DEALLOCATE(A, B, STAT = ALLOC_ERR)
```

The following shows another example:

```
INTEGER, ALLOCATABLE :: dataset(:,:,:)

INTEGER reactor, level, points, error

DATA reactor, level, points / 10, 50, 10 /

ALLOCATE (dataset(1:reactor,1:level,1:points), STAT = error)

DEALLOCATE (dataset, STAT = error)
```

See Also

- C to D
- ALLOCATE
- NULLIFY
- Arrays
- Dynamic Allocation

DECLARE and NODECLARE

General Compiler Directives: *DECLARE generates warnings for variables that have been used but have not been declared (like the IMPLICIT NONE statement). NODECLARE (the default) disables these warnings.*

Syntax

```
cDEC$ DECLARE

cDEC$ NODECLARE
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The DECLARE directive is primarily a debugging tool that locates variables that have not been properly initialized, or that have been defined but never used.

See Also

- C to D
- IMPLICIT
- General Compiler Directives

Building Applications: Compiler Directives Related to Options

DECODE

Statement: *Translates data from character to internal form. It is comparable to using internal files in formatted sequential READ statements.*

Syntax

```
DECODE (c,f,b[, IOSTAT=i-var] [, ERR=label]) [io-list]
```

c	Is a scalar integer expression. It is the number of characters to be translated to internal form.
f	Is a format identifier. An error occurs if more than one record is specified.
b	Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression. *b* contains the characters to be translated to internal form.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see I/O Status Specifier).
label	Is the label of an executable statement that receives control if an error occurs.
io-list	Is an I/O list. An I/O list is either an implied-DO list or a simple list of variables (except for assumed-size arrays). The list receives the data after translation to internal form. The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

The number of characters that the DECODE statement can translate depends on the data type of *b*. For example, an INTEGER(2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Example

In the following example, the DECODE statement translates the 12 characters in A to integer form (as specified by the FORMAT statement):

```
      DIMENSION K(3)
      CHARACTER*12 A,B
      DATA A/'123456789012'/
      DECODE(12,100,A) K
100   FORMAT(3I4)
      ENCODE(12,100,B) K(3), K(2), K(1)
```

The 12 characters are stored in array K:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

See Also

- C to D
- READ
- WRITE
- ENCODE

DEFAULT Clause

Parallel Directive Clause: *Lets you specify a scope for all variables in the lexical extent of a parallel region.*

Syntax

```
DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE )
```

The specifications have the following effects:

- PRIVATE - Makes all named objects in the lexical extent of the parallel region, including common block variables but excluding THREADPRIVATE variables, private to a thread as if you explicitly listed each variable in a PRIVATE clause.
- FIRSTPRIVATE - Makes all variables in the construct that have implicitly determined data-sharing attributes firstprivate as if you explicitly listed each variable in a FIRSTPRIVATE clause.
- SHARED - Makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if you explicitly listed each variable in a SHARED clause. If you do not specify a DEFAULT clause, this is the default.
- NONE - Specifies that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, you must specify the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION property of each variable you use in the lexical extent of the parallel region.

You can specify only one DEFAULT clause in a PARALLEL directive. You can exclude variables from a defined default by using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clauses.

Variables in THREADPRIVATE common blocks are not affected by this clause.

See Also

- C to D

Optimizing Applications: DEFAULT Clause

DEFINE and UNDEFINE

General Compiler Directives: *DEFINE creates a symbolic variable whose existence or value can be tested during conditional compilation. UNDEFINE removes a defined symbol.*

Syntax

```
cDEC$ DEFINE name[ = val]
cDEC$ UNDEFINE name
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
name	Is the name of the variable.
val	INTEGER(4). The value assigned to *name*.

DEFINE creates and UNDEFINE removes symbols for use with the IF (or IF DEFINED) compiler directive. Symbols defined with DEFINE directive are local to the directive. They cannot be declared in the Fortran program.

Because Fortran programs cannot access the named variables, the names can duplicate Fortran keywords, intrinsic functions, or user-defined names without conflict.

To test whether a symbol has been defined, use the IF DEFINED (*name*) directive. You can assign an integer value to a defined symbol. To test the assigned value of *name*, use the IF directive. IF test expressions can contain most logical and arithmetic operators.

Attempting to undefine a symbol that has not been defined produces a compiler warning.

The DEFINE and UNDEFINE directives can appear anywhere in a program, enabling and disabling symbol definitions.

Example

```
!DEC$ DEFINE testflag
!DEC$ IF DEFINED (testflag)
   write (*,*) 'Compiling first line'
!DEC$ ELSE
   write (*,*) 'Compiling second line'
!DEC$ ENDIF
!DEC$ UNDEFINE testflag
```

See Also

- C to D
- T to Z
- IF Directive Construct
- General Compiler Directives
- D compiler option

Building Applications: Compiler Directives Related to Options

Building Applications: Using Predefined Preprocessor Symbols

DEFINE FILE

Statement: *Establishes the size and structure of files with relative organization and associates them with a logical unit number.*

Syntax

```
DEFINE FILE u(m,n,U,asv) [,u(m,n,U,asv)] ...
```

u	Is a scalar integer constant or variable that specifies the logical unit number.
m	Is a scalar integer constant or variable that specifies the number of records in the file.
n	Is a scalar integer constant or variable that specifies the length of each record in 16-bit words (2 bytes).
U	Specifies that the file is unformatted (binary); this is the only acceptable entry in this position.

asv	Is a scalar integer variable, called the associated variable of the file. At the end of each direct access I/O operation, the record number of the next higher numbered record in the file is assigned to *asv*; *asv* must not be a dummy argument.

The DEFINE FILE statement is comparable to the OPEN statement. In situations where you can use the OPENstatement, OPEN is the preferable mechanism for creating and opening files.

The DEFINE FILE statement specifies that a file containing *m* fixed-length records, each composed of *n*16-bit words, exists (or will exist) on the specified logical unit. The records in the file are numbered sequentially from 1 through *m*.

A DEFINE FILE statement does not itself open a file. However, the statement must be executed before the first direct access I/O statement referring to the specified file. The file is opened when the I/O statement is executed.

If this I/O statement is a WRITE statement, a direct access sequential file is opened, or created if necessary.

If the I/O statement is a READ or FIND statement, an existing file is opened, unless the specified file does not exist. If a file does not exist, an error occurs.

The DEFINE FILE statement establishes the variable *asv* as the associated variable of a file. At the end of each direct access I/O operation, the Fortran I/O system places in *asv* the record number of the record immediately following the one just read or written.

The associated variable always points to the next sequential record in the file (unless the associated variable is redefined by an assignment, input, or FIND statement). So, direct access I/O statements can perform sequential processing on the file by using the associated variable of the file as the record number specifier.

Example

```
DEFINE FILE 3(1000,48,U,NREC)
```

In this example, the DEFINE FILE statement specifies that the logical unit 3 is to be connected to a file of 1000 fixed-length records; each record is forty-eight 16-bit words long. The records are numbered sequentially from 1 through 1000 and are unformatted.

After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the record just processed.

See Also

- C to D
- OPEN

DELDIRQQ

Portability Function: *Deletes a specified directory.*

Module

USE IFPORT

Syntax

```
result = DELDIRQQ (dir)
```

dir	(Input) Character*(*). String containing the path of the directory to be deleted.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The directory to be deleted must be empty. It cannot be the current directory, the root directory, or a directory currently in use by another process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example for GETDRIVEDIRQQ.

See Also

- C to D
- GETDRIVEDIRQQ
- GETDRIVEDIRQQ
- MAKEDIRQQ
- CHANGEDIRQQ
- CHANGEDRIVEQQ
- UNLINK

DELETE

Statement: *Deletes a record from a relative file.*

Syntax

```
DELETE ([UNIT=] io-unit, REC= r[, ERR= label] [, IOSTAT=i-var])
```

io-unit	Is an external unit specifier.
r	Is a scalar numeric expression indicating the record number to be deleted.
label	Is the label of the branch target statement that receives control if an error occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

In a relative file, the DELETE statement deletes the direct access record specified by *r*. If REC= *r* is omitted, the current record is deleted. When the direct access record is deleted, any associated variable is set to the next record number.

The DELETE statement logically removes the appropriate record from the specified file by locating the record and marking it as a deleted record. It then frees the position formerly occupied by the deleted record so that a new record can be written at that position.

NOTE. You must use compiler option vms for READs to detect that a record has been deleted.

Example

The following statement deletes the fifth record in the file connected to I/O unit 10:

```
DELETE (10, REC=5)
```

Suppose the following statement is specified:

```
DELETE (UNIT=9, REC=10, IOSTAT=IOS, ERR=20)
```

The tenth record in the file connected to unit 9 is deleted. If an error occurs, control is transferred to the statement labeled 20, and a positive integer is stored in the variable IOS.

See Also

- C to D

- Data Transfer I/O Statements
- Branch Specifiers
- vms compiler option

DELETEMENUQQ (W*32, W*64)

QuickWin Function: *Deletes a menu item from a QuickWin menu.*

Module

USE IFQWIN

Syntax

```
result = DELETEMENUQQ (menuID, itemID)
```

menuID	(Input) INTEGER(4). Identifies the menu that contains the menu item to be deleted, starting with 1 as the leftmost menu.
itemID	(Input) INTEGER(4). Identifies the menu item to be deleted, starting with 0 as the top menu item.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
LOGICAL(4) result
CHARACTER(25) str
str = 'Add to EDIT Menu'C   ! Append to 2nd menu
result = APPENDMENUQQ(2, $MENUENABLED, str, WINSTATUS)
! Delete third item (EXIT) from menu 1 (FILE)
result = DELETEMENUQQ(1, 3)
! Delete entire fifth menu (WINDOW)
result = DELETEMENUQQ(5,0)
END
```

See Also

- C to D
- APPENDMENUQQ
- INSERTMENUQQ
- MODIFYMENUFLAGSQQ
- MODIFYMENUROUTINEQQ
- MODIFYMENUSTRINGQQ

Building Applications: Using QuickWin Overview

Building Applications: Program Control of Menus

DELFILESQQ

Portability Function: *Deletes all files matching the name specification, which can contain wildcards (* and ?).*

Module

USE IFPORT

Syntax

```
result = DELFILESQQ (files)
```

files	(Input) Character*(*). Files to be deleted. Can contain wildcards (* and ?).

Results

The result type is INTEGER(2). The result is the number of files deleted.

You can use wildcards to delete more than one file at a time. DELFILESQQ does not delete directories or system, hidden, or read-only files. Use this function with caution because it can delete many files at once. If a file is in use by another process (for example, if it is open in another process), it cannot be deleted.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
USE IFCORE
INTEGER(4) len, count
CHARACTER(80) file
CHARACTER(1) ch
WRITE(*,*) "Enter names of files to delete: "
len = GETSTRQQ(file)
IF (file(1:len) .EQ. '*.*') THEN
   WRITE(*,*) "Are you sure (Y/N)?"
   ch = GETCHARQQ()
   IF ((ch .NE. 'Y') .AND. (ch .NE. 'y')) STOP
END IF
count = DELFILESQQ(file)
WRITE(*,*) "Deleted ", count, " files."
END
```

See Also

- C to D

- FINDFILEQQ

TYPE Statement (Derived Types)

Statement: *Declares a variable to be a derived type. It specifies the name of the user-defined type and the types of its components.*

Syntax

```
TYPE [[,type-attr-spec-list] :: ] name
component-definition
   [component-definition]. . .
END TYPE [ name ]
```

type-attr-spec-list Is *access-spec* or BIND (C).

access-spec Is the PUBLIC or PRIVATE keyword. The keyword can only be specified if the derived-type definition is in the specification part of a module.

name Is the name of the derived data type. It must not be the same as the name of any intrinsic type, or the same as the name of a derived type that can be accessed from a module.

component-definition Is one or more type declaration statements defining the component of derived type.
The first component definition can be preceded by an optional PRIVATE or SEQUENCE statement. (Only one PRIVATE or SEQUENCE statement can appear in a given derived-type definition.)
If SEQUENCE is present, all derived types specified in component definitions must be sequence types.
A *component definition* takes the following form:
type[[, *attr*] ::] *component*[(*a-spec*)] [**char-len*] [*init-ex*]

type Is a type specifier. It can be an intrinsic type or a previously defined derived type. (If the POINTER attribute follows this specifier, the type can also be any accessible derived type, including the type being defined.)

attr	Is an optional POINTER attribute for a pointer component, or an optional DIMENSION or ALLOCATABLE attribute for an array component. You cannot specify both the ALLOCATABLE and POINTER attribute. If DIMENSION is specified, it can be followed by an array specification. Each attribute can only appear once in a given *component-definition*.
component	Is the name of the component being defined.
a-spec	Is an optional array specification, enclosed in parentheses. If POINTER or ALLOCATABLE is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression. If the array bounds are not specified here, they must be specified following the DIMENSION attribute.
char-len	Is an optional scalar integer literal constant; it must be preceded by an asterisk (*). This parameter can only be specified if the component is of type CHARACTER.
init-ex	Is an initialization expression, or for pointer components, => NULL( ). This is a Fortran 95 feature. If *init-ex* is specified, a double colon must appear in the component definition. The equals assignment symbol (=) can only be specified for nonpointer components. The initialization expression is evaluated in the scoping unit of the type definition.

Description

If a name is specified following the END TYPE statement, it must be the same name that follows TYPE in the derived type statement.

A derived type can be defined only once in a scoping unit. If the same derived-type name appears in a derived-type definition in another scoping unit, it is treated independently.

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.

Two data entities have the same type if they are both declared to be of the same derived type (the derived-type definition can be accessed from a module or a host scoping unit).

If the entities are in different scoping units, they can also have the same derived type if they are declared with reference to different derived-type definitions, and if both derived-type definitions have all of the following:

- The same name
- A SEQUENCE statement (they both have sequence type)
- Components that agree in name, order, and attributes; components cannot be private

If BIND (C) is specified, the following rules apply:

- The derived type cannot be a SEQUENCE type.
- The derived type must have type parameters.
- Each component of the derived type must be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Example

```
!   DERIVED.F90
!   Define a derived-type structure,
!   type variables, and assign values
    TYPE member
      INTEGER age
      CHARACTER (LEN = 20) name
    END TYPE member
    TYPE (member) :: george
    TYPE (member) :: ernie
    george     = member( 33, 'George Brown' )
    ernie%age  = 56
    ernie%name = 'Ernie Brown'
    WRITE (*,*) george
    WRITE (*,*) ernie
    END
```

The following shows another example of a derived type:

```
TYPE mem_name
  SEQUENCE
  CHARACTER (LEN = 20) lastn
  CHARACTER (LEN = 20) firstn
  CHARACTER (len = 3) cos   ! this works because COS is a component name
END TYPE mem_name
TYPE member
    TYPE (mem_name) :: name
    SEQUENCE
    INTEGER age
    CHARACTER (LEN = 20) specialty
END TYPE member
```

In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
 TYPE (pair)
   INTEGER i, j
 END TYPE
 TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

The following example shows how you can use derived-type objects as components of other derived-type objects:

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
END TYPE
```

Objects of these derived types can then be used within a third derived-type specification, such as:

```
TYPE employee_data
  TYPE (employee_name) :: name
  TYPE (employee_addr) :: addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

See Also

- C to D
- E to F

- T to Z
- DIMENSION
- MAP...END MAP
- PRIVATE
- PUBLIC
- RECORD
- SEQUENCE
- STRUCTURE...END STRUCTURE
- Derived Data Types
- Default Initialization
- Structure Components
- Structure Constructors

Building Applications: Handling User-Defined Types

DFLOAT

Elemental Intrinsic Function (Generic):
Converts an integer to double-precision real type.

Syntax

```
result = DFLOAT (a)
```

a (Input) Must be of type integer.

Results

The result type is double-precision real (by default, REAL(8) or REAL*8). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

Specific Name [1]	Argument Type	Result Type [2]
	INTEGER(1)	REAL(8)
DFLOTI	INTEGER(2)	REAL(8)
DFLOTJ	INTEGER(4)	REAL(8)
DFLOTK	INTEGER(8)	REAL(8)

Specific Name [1]	Argument Type	Result Type [2]

[1]These specific functions cannot be passed as actual arguments.

[2]The setting of compiler options specifying double size can affect DFLOAT.

Example

DFLOAT (-4) has the value -4.0.

See Also

- C to D
- REAL

DFLOATI, DFLOATJ, DFLOATK

Portability Functions: *Convert an integer to double-precision real type.*

Module

USE IFPORT

Syntax

```
result = DFLOATI (i)
result = DFLOATJ (j)
result = DFLOATK (k)
```

i	(Input) Must be of type INTEGER(2).
j	(Input) Must be of type INTEGER(4).
k	(Input) Must be of type INTEGER(8).

Results

The result type is double-precision real (REAL(8) or REAL*8).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- C to D
- DFLOAT

DIGITS

Inquiry Intrinsic Function (Generic): *Returns the number of significant digits for numbers of the same type and kind parameters as the argument.*

Syntax

```
result = DIGITS (x)
```

x	(Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of type default integer.

The result has the value q if x is of type integer; it has the value p if x is of type real. Integer parameter q is defined in Model for Integer Data; real parameter p is defined in Model for Real Data.

Example

If x is of type REAL(4), DIGITS(x) has the value 24.

See Also

- C to D
- EXPONENT
- RADIX
- FRACTION
- Data Representation Models

DIM

Elemental Intrinsic Function (Generic):
Returns the difference between two numbers (if the difference is positive).

Syntax

```
result = DIM (x, y)
```

x	(Input) Must be of type integer or real.
y	(Input) Must have the same type and kind parameters as *x*.

Results

The result type is the same as *x*. The value of the result is *x*- *y* if *x* is greater than *y*; otherwise, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Specific Name	Argument type	Result Type
BDIM	INTEGER(1)	INTEGER(1)
IIDIM[1]	INTEGER(2)	INTEGER(2)
IDIM [2]	INTEGER(4)	INTEGER(4)
KIDIM	INTEGER(8)	INTEGER(8)
DIM	REAL(4)	REAL(4)
DDIM	REAL(8)	REAL(8)
QDIM	REAL(16)	REAL(16)

[1]Or HDIM.

[2]Or JIDIM. For compatibility, IDIM can also be specified as a generic function for integer types.

Example

DIM (6, 2) has the value 4.

DIM (-4.0, 3.0) has the value 0.0.

The following shows another example:

```
INTEGER i
REAL r
REAL(8) d
i = IDIM(10, 5)            ! returns 5
r = DIM (-5.1, 3.7)        ! returns 0.0
d = DDIM (10.0D0, -5.0D0)  ! returns 15.0D0
```

See Also

- C to D
- Argument Keywords in Intrinsic Procedures

DIMENSION

Statement and Attribute: *Specifies that an object is an array, and defines the shape of the array.*

Syntax

The DIMENSION attribute can be specified in a type declaration statement or a DIMENSION statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] DIMENSION (*a-spec*) [, *att-ls*] :: *a*[(*a-spec*)][, *a*[(*a-spec*)]] ...

Statement:

DIMENSION [::]*a*(*a-spec*) [, *a*(*a-spec*)] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
a-spec	Is an array specification. It can be any of the following: • An explicit-shape specification; for example, a(10,10) • An assumed-shape specification; for example, a(:) • A deferred-shape specification; for example, a(:,:) • An assumed-size specification; for example, a(10,*)

For more information on array specifications, see Declaration Statements for Arrays.
In a type declaration statement, any array specification following an array overrides any array specification following DIMENSION.

a Is the name of the array being declared.

Description

The DIMENSION attribute allocates a number of storage elements to each array named, one storage element to each array element in each dimension. The size of each storage element is determined by the data type of the array.

The total number of storage elements assigned to an array is equal to the number produced by multiplying together the number of elements in each dimension in the array specification. For example, the following statement defines ARRAY as having 16 real elements of 4 bytes each and defines MATRIX as having 125 integer elements of 4 bytes each:

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

An array can also be declared in the following statements: ALLOCATABLE, AUTOMATIC, COMMON, POINTER, STATIC, TARGET.

Example

The following examples show type declaration statements specifying the DIMENSION attribute:

```
REAL, DIMENSION(10, 10) :: A, B, C(10, 15)  ! Specification following C
                                            ! overrides the one following
                                            ! DIMENSION

REAL(8), DIMENSION(5,-2:2) :: A,B,C
```

The following are examples of the DIMENSION statement:

```
DIMENSION BOTTOM(12,24,10)

DIMENSION X(5,5,5), Y(4,85), Z(100)

DIMENSION MARK(4,4,4,4)

SUBROUTINE APROC(A1,A2,N1,N2,N3)

DIMENSION A1(N1:N2), A2(N3:*)

CHARACTER(LEN = 20) D

DIMENSION A(15), B(15, 40), C(-5:8, 7), D(15)
```

You can also declare arrays by using type and ALLOCATABLE statements, for example:

```
INTEGER A(2,0:2)
COMPLEX F
ALLOCATABLE F(:,:)
REAL(8), ALLOCATABLE, DIMENSION( :, :, : ) :: E
```

You can specify both the upper and lower dimension bounds. If, for example, one array contains data from experiments numbered 28 through 112, you could dimension the array as follows:

```
DIMENSION experiment(28:112)
```

Then, to refer to the data from experiment 72, you would reference experiment(72).

Array elements are stored in column-major order: the leftmost subscript is incremented first when the array is mapped into contiguous memory addresses. For example, consider the following statements:

```
INTEGER(2) a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

These are equivalent to:

```
INTEGER(2) a
DIMENSION a(2, 0:2)
DATA a /1, 2, 3, 4, 5, 6/
```

If a is placed at location 1000 in memory, the preceding DATA statement produces the following mapping.

Array element	Address	Value
a(1,0)	1000	1
a(2,0)	1002	2
a(1,1)	1004	3
a(2,1)	1006	4
a(1,2)	1008	5
a(2,2)	100A	6

The following DIMENSION statement defines an assumed-size array in a subprogram:

```
DIMENSION data (19,*)
```

At execution time, the array data is given the size of the corresponding array in the calling program.

The following program fragment dimensions two arrays:

```
   ...
   SUBROUTINE Subr (matrix, rows, vector)
   REAL MATRIX, VECTOR
   INTEGER ROWS
   DIMENSION MATRIX (ROWS,*), VECTOR (10),
  +  LOCAL (2,4,8)
   MATRIX (1,1) = VECTOR (5)
   ...
```

See Also

- C to D
- ALLOCATE
- Declaration Statements for Arrays
- Arrays

DISPLAYCURSOR (W*32, W*64)

Graphics Function: *Controls cursor visibility.*

Module

USE IFQWIN

Syntax

`result = DISPLAYCURSOR (toggle)`

toggle	(Input) INTEGER(2). Constant that defines the cursor state. Has two possible values: • $GCURSOROFF - Makes the cursor invisible regardless of its current shape and mode.

- $GCURSORON - Makes the cursor always visible in graphics mode.

Results

The result type is INTEGER(2). The result is the previous value of *toggle*.

Cursor settings hold only for the currently active child window. You need to call DISPLAYCURSOR for each window in which you want the cursor to be visible.

A call to SETWINDOWCONFIG turns off the cursor.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- C to D
- SETTEXTCURSOR
- SETWINDOWCONFIG

DISTRIBUTE POINT

General Compiler Directive: *Specifies loop distribution.*

Syntax

*c*DEC$ DISTRIBUTE POINT

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

Loop distribution causes large loops to be distributed (split) into smaller ones. The resulting loops contain a subset of the instructions from the initial loop. Loop distribution can enable software pipelining to be applied to more loops. It can also reduce register pressure and improve both instruction and data cache use.

If the directive is placed before a loop, the compiler will determine where to distribute; data dependencies are observed.

If the directive is placed inside a loop, the distribution is performed after the directive and any loop-carried dependencies are ignored. Currently only one distribute directive is supported if the directive is placed inside the loop.

Example

```
!DEC$ DISTRIBUTE POINT
   do i =1, m
     b(i) = a(i) +1
     ....
     c(i) = a(i) + b(i) ! Compiler will decide
     ! where to distribute.
     ! Data dependencies are
     ! observed
     ....
     d(i) = c(i) + 1
   enddo
   do i =1, m
     b(i) = a(i) +1
     ....
!DEC$ DISTRIBUTE POINT
     call sub(a, n)! Distribution will start here,
     ! ignoring all loop-carried
     ! depedencies
     c(i) = a(i) + b(i)
     ....
     d(i) = c(i) + 1
   enddo
```

See Also

- C to D
- Rules for General Directives that Affect DO Loops

DLGEXIT (W*32, W*64)

Dialog Subroutine: *Closes an open dialog box.*

Module

USE IFLOGM

Syntax

CALL DLGEXIT (*dlg*)

dlg	(Input) Derived type dialog. Contains dialog box parameters. The components of the type dialog are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

If you want to exit a dialog box on a condition other than the user selecting the OK or Cancel button, you need to include a call to DLGEXIT from within your callback routine. DLGEXIT saves the data associated with the dialog box controls and then closes the dialog box. The dialog box is exited after DLGEXIT has returned control back to the dialog manager, not immediately after the call to DLGEXIT.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
SUBROUTINE EXITSUB (dlg, exit_button_id, callbacktype)
USE IFLOGM
TYPE (DIALOG) dlg
INTEGER exit_button_id, callbacktype
...
  CALL DLGEXIT (dlg)
```

See Also

- C to D
- DLGSETRETURN
- DLGINIT
- DLGMODAL

- DLGMODELESS

Building Applications: Setting Return Values and Exiting

DLGFLUSH (W*32, W*64)

Dialog Subroutine: *Updates the display of a dialog box.*

Module

USE IFLOGM

Syntax

`CALL DLGFLUSH (dlg[,flushall])`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
flushall	(Input; optional) Logical. If .FALSE. (the default), then only the controls that the dialog routines have marked as changed are updated. If .TRUE., all controls are updated with the state of the controls as known by the dialog routines. Normally, you would not set *flushall* to .TRUE..

When your application calls DLGSET to change a property of a control in a dialog box, the change is not immediately reflected in the displayed dialog box. Changes are applied when the dialog box is first displayed, and then after every dialog callback to the user's code.

This design expects that, after a call to DLGMODAL or DLGMODELESS, every call to DLGSET will be made from within a callback routine, and that the callback routine finishes quickly. This is true most of the time.

However, there may be cases where you want to change a control outside of a dialog callback, or from within a loop in a dialog callback.

In these cases, DLGFLUSH is required, but is not always sufficient, to update the dialog display. DLGFLUSH sends pending Windows* system messages to the dialog box and the controls that it contains. However, many display changes do not appear until after the program reads and processes these messages. A loop that processes the pending messages may be required; for example:

```
use IFWINTY
use USER32
use IFLOGM
logical lNotQuit, lret
integer iret
TYPE (T_MSG) mesg
lNotQuit = .TRUE.
do while (lNotQuit .AND. (PeekMessage(mesg, 0, 0, 0, PM_NOREMOVE) <> 0))
  lNotQuit = GetMessage(mesg, NULL, 0, 0)
    if (lNotQuit) then
      if (DLGISDLGMESSAGE(mesg) .EQV. .FALSE) then
         lret = TranslateMessage(mesg)
         iret = DispatchMessage(mesg)
      end if
    end if
end do
```

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- C to D
- DLGINIT
- DLGMODAL
- DLGMODELESS
- DLGSET
- DLGSETSUB

DLGGET, DLGGETINT, DLGGETLOG, DLGGETCHAR (W*32, W*64)

Dialog Functions: *Return the state of the dialog control variable.*

Module

USE IFLOGM

Syntax

```
result = DLGGET (dlg,controlid,value[,index])
result = DLGGETINT (dlg,controlid,value[,index])
result = DLGGETLOG (dlg,controlid,value[,index])
result = DLGGETCHAR (dlg,controlid,value[,index])
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
value	(Output) Integer, logical, or character. The value of the control's variable.
index	(Input; optional) Integer. Specifies the control variable whose value is retrieved. Necessary if the control has more than one variable of the same data type and you do not want to get the value of the default for that type.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the DLGGET functions to retrieve the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variable associated with it, but not necessarily all. The control variables are listed in the table in *Building Applications*: Control Indexes. The types of controls they are associated with are listed in the table in *Building Applications*: Available Indexes for Each Dialog Control.

You can use DLGGET to retrieve the value of any variable. You can also use DLGGETINT to retrieve an integer value, or DLGGETLOG and DLGGETCHAR to retrieve logical and character values, respectively. If you use DLGGET, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to retrieve a variable type that is not available, the DLGGET functions return .FALSE..

If two or more controls have the same *controlid*, you cannot use these controls in a DLGGET operation. In this case the function returns .FALSE..

The dialog box does not need to be open to access its control variables.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

INCLUDE "THISDLG.FD"

TYPE (DIALOG)  dlg

INTEGER        val

LOGICAL        retlog, is_checked

CHARACTER(256) text

...

retlog = DLGGET (dlg, IDC_CHECKBOX1, is_checked, dlg_status)

retlog = DLGGET (dlg, IDC_SCROLLBAR2, val, dlg_range)

retlog = DLGGET (dlg, IDC_STATIC_TEXT1, text, dlg_title)

...
```

See Also

- C to D
- DLGSET
- DLGSETSUB
- DLGINIT
- DLGMODAL
- DLGMODELESS

Building Applications: Using Dialogs for Applications Control Overview

Building Applications: Dialog Routines

Building Applications: Using Dialog Controls Overview

Building Applications: Specifying Control Indexes

Building Applications: Using Check Boxes and Radio Buttons

Building Applications: Using Edit Boxes

Building Applications: Using Scroll Bars

DLGINIT, DLGINITWITHRESOURCEHANDLE (W*32, W*64)

Dialog Functions: *Initialize a dialog box.*

Module

USE IFLOGM

Syntax

```
result = DLGINIT (id,dlg)
```

```
result = DLGINITWITHRESOURCEHANDLE (id,hinst,dlg)
```

id	(Input) INTEGER(4). Dialog identifier. Can be either the symbolic name for the dialog or the identifier number, both listed in the Include file (with extension .FD).
dlg	(Output) Derived type `dialog`. Contains dialog box parameters.
hinst	(Input) INTEGER(4). Module instance handle in which the dialog resource can be found.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

DLGINIT must be called to initialize a dialog box before it can be used with DLGMODAL, DLGMODELESS, or any other dialog function.

DLGINIT will only search for the dialog box resource in the main application. For example, it will not find a dialog box resource that has been built into a dynamic link library.

DLGINITWITHRESOURCEHANDLE can be used when the dialog resource is not in the main application. If the dialog resource is in a dynamic link library (DLL), *hinst* must be the value passed as the first argument to the DLLMAIN procedure.

Dialogs can be used from any application, including console, QuickWin, and Windows* applications.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

INCLUDE 'DLG1.FD'

LOGICAL retlog

TYPE (DIALOG) thisdlg

...

retlog = DLGINIT (IDD_DLG3, thisdlg)

IF (.not. retlog) THEN

  WRITE (*,*) 'ERROR: dialog not found'

ELSE

...
```

See Also

- C to D
- DLGEXIT
- DLGMODAL
- DLGMODELESS
- DLGUNINIT

Building Applications: Initializing and Activating the Dialog Bo

DLGISDLGMESSAGE, DLGISDLGMESSAGEWITHDLG (W*32, W*64)

Dialog Functions: *Determine whether the specified message is intended for one of the currently displayed modeless dialog boxes, or a specific dialog box.*

Module

USE IFLOGM

Syntax

```
result = DLGISDLGMESSAGE (mesg)
```

```
result = DLGISDLGMESSAGEWITHDLG (mesg, dlg)
```

mesg	(Input) Derived type `T_MSG`. Contains a Windows message.
dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

Results

The result type is LOGICAL(4). The result is .TRUE. if the message is processed by the dialog box. Otherwise, the result is .FALSE. and the message should be further processed.

DLGISDLGMESSAGE must be called in the message loop of Windows applications that display a modeless dialog box using DLGMODELESS. DLGISDGMESSAGE determines whether the message is intended for one of the currently displayed modeless dialog boxes. If it is, it passes the message to the dialog box to be processed.

DLGISDLGMESSAGEWITHDLG specifies a particular dialog box to check. Use DLGISDLGMESSAGEWITHDLG when the message loop is in a main application and the currently active modeless dialog box was created by a DLL.

Compatibility

WINDOWS

Example

```
use IFLOGM
include 'resource.fd'
type (DIALOG) dlg
type (T_MSG)  mesg
integer*4 ret
logical*4 lret
...
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
...
lret = DlgModeless(dlg, nCmdShow)
...
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
      lret = TranslateMessage( mesg )
      ret  = DispatchMessage( mesg )
   end if
end do
! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg%wParam
return
```

See Also

- C to D
- DLGMODELESS

Building Applications: Using a Modeless Dialog Box

DLGMODAL, DLGMODALWITHPARENT (W*32, W*64)

Dialog Functions: *Display a dialog box and process user control selections made within the box.*

Module

USE IFLOGM

Syntax

```
result = DLGMODAL (dlg)
result = DLGMODAL (dlg, hwndParent)
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
hwndParent	(Input) Integer. Specifies the parent window for the dialog box. If omitted, the value is determined in this order: **1.** If DLGMODAL is called from the callback of a modal or modeless dialog box, then that dialog box is the parent window. **2.** If it is a QuickWin or Standard Graphics application, then the frame window is the parent window. **3.** The Windows* desktop window is the parent window.

Results

The result type is INTEGER(4). By default, if successful, it returns the identifier of the control that caused the dialog to exit; otherwise, it returns -1. The return value can be changed with the DLGSETRETURN subroutine.

During execution, DLGMODAL displays a dialog box and then waits for user control selections. When a control selection is made, the callback routine, if any, of the selected control (set with DLGSETSUB) is called.

The dialog remains active until an exit control is executed: either the default exit associated with the OK and Cancel buttons, or DLGEXIT within your own control callbacks. DLGMODAL does not return a value until the dialog box is exited.

The default return value for DLGMODAL is the identifier of the control that caused it to exit (for example, IDOK for the OK button and IDCANCEL for the Cancel button). You can specify your own return value with DLGSETRETURN from within one of your dialog control callback routines. You should not specify -1 as your return value, because this is the error value DLGMODAL returns if it cannot open the dialog.

Use DLGMODALWITHPARENT when you want the parent window to be other than the default value (see argument *hwndParent* above). In particular, in an SDI or MDI Windows application, you may want the parent window to be the main application window. The parent window is disabled for user input while the modal dialog box is displayed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

INCLUDE "MYDLG.FD"

INTEGER return

TYPE (DIALOG) mydialog

...

return = DLGMODAL (mydialog)

...
```

See Also

- C to D
- DLGSETRETURN
- DLGSETSUB
- DLGINIT
- DLGEXIT

Building Applications: Dialog Callback Routines

Building Applications: Initializing and Activating the Dialog Box

Building Applications: Setting Return Values and Exiting

DLGMODELESS (W*32, W*64)

Dialog Function: *Displays a modeless dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGMODELESS (dlg[,nCmdShow,hwndParent])`

dlg (Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user. The variable passed to this function must remain in memory for the duration of the dialog box, that is from the DLGINIT call through the DLGUNINIT call.
The variable can be declared as global data in a module, as a variable with the STATIC attribute, or in a calling procedure that is active for the duration of the dialog box. It must not be an AUTOMATIC variable in the procedure that calls DLGMODELESS.

nCmdShow (Input) Integer. Specifies how the dialog box is to be shown. It must be one of the following values:

Value	Description
SW_HIDE	Hides the dialog box.
SW_MINIMIZE	Minimizes the dialog box.
SW_RESTORE	Activates and displays the dialog box. If the dialog box is minimized or maximized, the Windows system restores it to its original size and position.

Value	Description
SW_SHOW	Activates the dialog box and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates the dialog box and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the dialog box and displays it as an icon.
SW_SHOWMINNOACTIVE	Displays the dialog box as an icon. The window that is currently active remains active.
SW_SHOWNA	Displays the dialog box in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays the dialog box in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays the dialog box. If the dialog box is minimized or maximized, the Windows system restores it to its original size and position.

The default value is SW_SHOWNORMAL.

nCmdShow (Input) Integer. Specifies the parent window for the dialog box. The default value is determined in this order:

1. If DLGMODELESS is called from a callback of a modeless dialog box, then that dialog box is the parent window.

2. The Windows desktop window is the parent window.

Results

The result type is LOGICAL(4). The value is .TRUE. if the function successfully displays the dialog box. Otherwise the result is .FALSE..

During execution, DLGMODELESS displays a modeless dialog box and returns control to the calling application. The dialog box remains active until DLGEXIT is called, either explicitly or as the result of the invocation of a default button callback.

DLGMODELESS is typically used in a Windows application. The application must contain a message loop that processes Windows messages. The message loop must call DLGISDLGMESSAGE for each message. See the example below in the Example section. Multiple modeless dialog boxes can be displayed at the same time. A modal dialog box can be displayed from a modeless dialog box by calling DLGMODAL from a modeless dialog callback. However, DLGMODELESS cannot be called from a modal dialog box callback.

DLGMODELESS also can be used in a Console, DLL, or LIB project. However, the requirements remain that the application must contain a message loop and must call DLGISDLGMESSAGE for each message. For an example of calling DLGMODELESS in a DLL project, see the Dllprgrs sample in the `...\SAMPLES\DIALOG`folder.

Use the DLG_INIT callback with DLGSETSUB to perform processing immediately after the dialog box is created and before it is displayed, and to perform processing immediately before the dialog box is destroyed.

Compatibility

WINDOWS CONSOLE DLL LIB

Example

```
use IFLOGM
include 'resource.fd'
type (DIALOG)   dlg
type (T_MSG)    mesg
integer*4   ret
logical*4   lret
...
! Create the main dialog box and set up the controls and callbacks
lret = DlgInit(IDD_THERM_DIALOG, dlg)
lret = DlgSetSub(dlg, IDD_THERM_DIALOG, ThermSub)
...
lret = DlgModeless(dlg, nCmdShow)
...
! Read and process messsages
do while( GetMessage (mesg, NULL, 0, 0) )
   ! Note that DlgIsDlgMessage must be called in order to give
   ! the dialog box first chance at the message.
   if ( DlgIsDlgMessage(mesg) .EQV. .FALSE. ) then
     lret = TranslateMessage( mesg )
     ret  = DispatchMessage( mesg )
   end if
end do
   ! Cleanup dialog box memory and exit the application
call DlgUninit(dlg)
WinMain = mesg%wParam
return
```

See Also

- C to D
- DLGSETSUB
- DLGINIT
- DLGEXIT
- DLGISDLGMESSAGE

Building Applications: Using a Modeless Dialog Box

DLGSENDCTRLMESSAGE (W*32, W*64)

Dialog Function: *Sends a Windows message to a dialog box control.*

Module

USE IFLOGM

Syntax

`result = DLGSENDCTRLMESSAGE (dlg,controlid,msg,wparam,lparam)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of the control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
msg	(Input) Integer. Derived type `T_MSG`. Specifies the message to be sent.
wparam	(Input) Integer. Specifies additional message specific information.
lparam	(Input) Integer. Specifies additional message specific information.

Results

The result type is INTEGER(4). The value specifies the result of the message processing and depends upon the message sent.

The dialog box must be currently active by a call to DLGMODAL or DLGMODELESS. This function does not return until the message has been processed by the control.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFLOGM
include 'resource.fd'
type (dialog)    dlg
integer          callbacktype
integer          cref
integer          iret
if (callbacktype == dlg_init) then
    ! Change the color of the Progress bar to red
    ! NOTE:  The following message succeeds only if Internet Explorer 4.0
    !        or later is installed
        cref = Z'FF'    ! Red
        iret = DlgSendCtrlMessage(dlg, IDC_PROGRESS1, PBM_SETBARCOLOR, 0, cref)
endif
```

See Also

- C to D
- DLGINIT
- DLGSETSUB
- DLGMODAL
- DLGMODELESS

DLGSET, DLGSETINT, DLGSETLOG, DLGSETCHAR (W*32, W*64)

Dialog Functions: *Set the values of dialog control variables.*

Module

USE IFLOGM

Syntax

```
result = DLGSET (dlg,controlid,value[,index])
result = DLGSETINT (dlg,controlid,value[,index])
result = DLGSETLOG (dlg,controlid,value[,index])
result = DLGSETCHAR (dlg,controlid,value[,index])
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be either the symbolic name for the control or the identifier number, both listed in the Include file (with extension .FD).
value	(Input) Integer, logical, or character. The value of the control's variable.
index	(Input; optional) Integer. Specifies the control variable whose value is set. Necessary if the control has more than one variable of the same data type and you do not want to set the value of the default for that type.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, the result is .FALSE..

Use the DLGSET functions to set the values of variables associated with your dialog box controls. Each control has at least one of the integer, logical, or character variables associated with it, but not necessarily all. The control variables are listed in the table in *Building Applications*: Control Indexes. The types of controls they are associated with are listed in the table in *Building Applications*: Available Indexes for Each Dialog Control.

You can use DLGSET to set any control variable. You can also use DLGSETINT to set an integer variable, or DLGSETLOG and DLGSETCHAR to set logical and character values, respectively. If you use DLGSET, you do not have to worry about matching the function to the variable type. If you use the wrong function type for a variable or try to set a variable type that is not available, the DLGSET functions return .FALSE..

Calling DLGSET does not cause a callback routine to be called for the changing value of a control. In particular, when inside a callback, performing a DLGSET on a control does not cause the associated callback for that control to be called. Callbacks are invoked automatically only by user action on the controls in the dialog box. If the callback routine needs to be called, you can call it manually after the DLGSET is executed.

If two or more controls have the same *controlid*, you cannot use these controls in a DLGSET operation. In this case the function returns .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM
INCLUDE "DLGRADAR.FD"
TYPE (DIALOG)  dlg
LOGICAL        retlog
...
retlog = DLGSET (dlg, IDC_SCROLLBAR1, 400, dlg_range)
retlog = DLGSET (dlg, IDC_CHECKBOX1, .FALSE., dlg_status)
retlog = DLGSET (dlg, IDC_RADIOBUTTON1, "Hot Button", dlg_title)
...
```

See Also

- C to D
- DLGSETSUB
- DLGGET

Building Applications: Using Dialogs for Applications Control Overview

Building Applications: Dialog Routines

Building Applications: Using Dialog Controls Overview

Building Applications: Specifying Control Indexes

Building Applications: Using Check Boxes and Radio Buttons

Building Applications: Using Edit Boxes

Building Applications: Using Scroll Bars

DLGSETCTRLEVENTHANDLER (W*32, W*64)

Dialog Function: *Assigns user-written event handlers to ActiveX controls in a dialog box.*

Module

USE IFLOGM

Syntax

`result = DLGSETCTRLEVENTHANDLER (dlg,controlid,handler,dispid[,iid])`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be the symbolic name for the control or the identifier number, both listed in the include (with extension .FD) file.
handler	(Input) Name of the routine to be called when the event occurs. It must be declared EXTERNAL.
dispid	(Input) Integer. Specifies the member id of the method in the event interface that identifies the event.
iid	(Input; optional) Derived type `GUID`, which is defined in the IFWINTY module. Specifies the interface identifier of the source (event) interface. If omitted, the default source interface of the ActiveX control is used.

Results

The result type is INTEGER(4). The result is an HRESULT describing the status of the operation.

When the ActiveX control event occurs, the handler associated with the event is called. You call DLGSETCTRLEVENTHANDLER to specify the handler to be called.

The events supported by an ActiveX control and the interfaces of the handlers are determined by the ActiveX control.

You can find this information in one of the following ways:

- By reading the documentation of the ActiveX control.

- By using a tool that lets you examine the type information of the ActiveX control;, such as the OLE-COM Object Viewer.
- By using the Fortran Module Wizard to generate a module that contains Fortran interfaces to the ActiveX control, and examining the generated module.

The handler that you define in your application must have the interface that the ActiveX control expects, including calling convention and parameter passing mechanisms. Otherwise, your application will likely crash in unexpected ways because of the application's stack getting corrupted.

Note that an object is always the first parameter in an event handler. This object value is a pointer to the control's source (event) interface, *not* the IDispatch pointer of the control. You can use DLGGET with the DLG_IDISPATCH index to retrieve the control's IDispatch pointer.

For more information, see *Building Applications*: Using ActiveX Controls Overview.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

ret = DlgSetCtrlEventHandler(            &
      dlg,                          &
      IDC_ACTIVEMOVIECONTROL1,      & ! Identifies the control
      ReadyStateChange,             & ! Name of the event handling routine
      -609,                         & ! Member id of the ActiveMovie's
                                    & !   control ReadyStateChange event.
      IID_DActiveMovieEvents2 )          ! Identifer of the source (event)
                                         !    interface.
```

See Also

- C to D
- DLGINIT
- DLGGET
- DLGMODAL
- DLGMODELESS

- DLGSETSUB

DLGSETRETURN (W*32, W*64)

Dialog Subroutine: *Sets the return value for the DLGMODAL function from within a callback subroutine.*

Module

USE IFLOGM

Syntax

`CALL DLGSETRETURN (dlg,retval)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
retval	(Input) Integer. Specifies the return value for DLGMODAL upon exiting.

DLGSETRETURN overrides the default return value with *retval*. You can set your own value as a means of determining the condition under which the dialog box was closed. The default return value for an error condition is -1, so you should not use -1 as your return value.

DLGSETRETURN should be called from within a callback routine, and is generally used with DLGEXIT, which causes the dialog box to be exited from a control callback rather than the user selecting the OK or Cancel button.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
SUBROUTINE SETRETSUB (dlg, button_id, callbacktype)
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (DIALOG) dlg
LOGICAL     is_checked, retlog
INTEGER     return, button_id, callbacktype
...
retlog = DLGGET(dlg, IDC_CHECKBOX4, is_checked, dlg_state)
IF (is_checked) THEN
   return = 999
ELSE
   return = -999
END IF
CALL DLGSETRETURN (dlg, return)
CALL DLGEXIT (dlg)
END SUBROUTINE SETRETSUB
```

See Also

- C to D
- DLGEXIT
- DLGMODAL

Building Applications: Setting Return Values and Exiting

DLGSETSUB (W*32, W*64)

Dialog Function: *Assigns your own callback subroutines to dialog controls and to the dialog box.*

Module

USE IFLOGM

Syntax

```
result = DLGSETSUB (dlg,controlid,value[,index])
```

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
controlid	(Input) Integer. Specifies the identifier of a control within the dialog box. Can be the symbolic name for the control or the identifier number, both listed in the include (with extension .FD) file, or it can be the identifier of the dialog box.
value	(Input) EXTERNAL. Name of the routine to be called when the callback event occurs.
index	(Input; optional) Integer. Specifies which callback routine is executed when the callback event occurs. Necessary if the control has more than one callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

When a callback event occurs (for example, when you select a check box), the callback routine associated with that callback event is called. You use DLGSETSUB to specify the subroutine to be called. All callback routines should have the following interface:

SUBROUTINE *callbackname*(*dlg*, *control_id*, *callbacktype*)

!DEC$ ATTRIBUTES DEFAULT :: *callbackname*

callbackname	Is the name of the callback routine.
dlg	Refers to the dialog box and allows the callback to change values of the dialog controls.
control_id	Is the name of the control that caused the callback.
callbacktype	(Input; optional) Integer. Specifies which callback routine is executed when the callback event occurs. Necessary if the control has more than one callback routine.

The *control_id* and *callbacktype* parameters let you write a single subroutine that can be used with multiple callbacks from more than one control. Typically, you do this for controls comprising a logical group. You can also associate more than one callback routine with the same control, but you must use then use *index* parameter to indicate which callback routine to use.

The *control_id* can also be the identifier of the dialog box. The dialog box supports two *callbacktype* s, DLG_INIT and DLG_SIZECHANGE. The DLG_INIT callback is executed immediately after the dialog box is created with *callbacktype*DLG_INIT, and immediately before the dialog box is destroyed with *callbacktype*DLG_DESTROY. DLG_SIZECHANGE is called when the size of a dialog is changed.

Callback routines for a control are called after the value of the control has been updated based on the user's action.

If two or more controls have the same *controlid*, you cannot use these controls in a DLGSETSUB operation. In this case, the function returns .FALSE..

For more information, see *Building Applications*: Dialog Callback Routines.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
PROGRAM DLGPROG
USE IFLOGM
INCLUDE "MYDLG.FD"
TYPE (dialog) mydialog
LOGICAL retlog
INTEGER return
EXTERNAL RADIOSUB
retlog = DLGINIT(IDD_mydlg, dlg)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON1, RADIOSUB)
retlog = DLGSETSUB (mydialog, IDC_RADIO_BUTTON2, RADIOSUB)
return = DLGMODAL(dlg)
END
SUBROUTINE RADIOSUB( dlg, id, callbacktype )
!DEC$ ATTRIBUTES DEFAULT :: callbackname
  USE IFLOGM
  TYPE (dialog) dlg
  INTEGER id, callbacktype
  INCLUDE 'MYDLG.FD'
  CHARACTER(256) text
  INTEGER cel, far, retint
  LOGICAL retlog
  SELECT CASE (id)
    CASE (IDC_RADIO_BUTTON1)
    ! Radio button 1 selected by user so
    ! change text accordingly
      text = 'Statistics Package A'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
```

```
    CASE (IDC_RADIO_BUTTON2)
    ! Radio button 2 selected by user so
    ! change text accordingly
      text = 'Statistics Package B'
      retlog = DLGSET( dlg, IDC_STATICTEXT1, text )
  END SELECT
END SUBROUTINE RADIOSUB
```

See Also

- C to D
- DLGSET
- DLGGET

Building Applications: Initializing and Activating the Dialog Box

DLGSETTITLE (W*32, W*64)

Dialog Subroutine: *Sets the title of a dialog box.*

Module

USE IFLOGM

Syntax

`CALL DLGSETTITLE (dlg,title)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.
title	(Input) Character*(*). Specifies text to be the title of the dialog box.

Use this routine when you want to specify the title for a dialog box.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

INCLUDE "MYDLG.FD"

TYPE (DIALOG) mydialog

LOGICAL retlog

...

retlog = DLGINIT(IDD_mydlg, mydialog)

...

CALL DLGSETTITLE(mydialog, "New Title")

...
```

See Also

- C to D
- DLGINIT
- DLGMODAL
- DLGMODELESS

DLGUNINIT (W*32, W*64)

Dialog Subroutine: *Deallocates memory associated with an initialized dialog.*

Module

USE IFLOGM

Syntax

`CALL DLGUNINIT (dlg)`

dlg	(Input) Derived type `dialog`. Contains dialog box parameters. The components of the type `dialog` are defined with the PRIVATE attribute, and cannot be changed or individually accessed by the user.

You should call DLGUNINIT when a dialog that was successfully initialized by DLGINIT is no longer needed. DLGUNINIT should only be called on a dialog initialized with DLGINIT. If it is called on an uninitialized dialog or one that has already been deallocated with DLGUNINIT, the result is undefined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFLOGM

INCLUDE "MYDLG.FD"

TYPE (DIALOG)  mydialog

LOGICAL        retlog

...

retlog = DLGINIT(IDD_mydlg, mydialog)

...

CALL DLGUNINIT (mydialog)

END
```

See Also

- C to D
- DLGINIT
- DLGMODAL
- DLGMODELESS
- DLGEXIT

Building Applications: Initializing and Activating the Dialog Box

DNUM

Elemental Intrinsic Function (Specific):
Converts a character string to a REAL(8) value. This function cannot be passed as an actual argument.

Syntax

```
result = DNUM (i)
```

i	(Input) Must be of type character.

Results

The result type is REAL(8). The result value is the double-precision real value represented by the character string *i*.

Example

DNUM ("3.14159") has the value 3.14159 of type REAL(8).

The following sets x to 311.0:

```
CHARACTER(3) i
DOUBLE PRECISION x
i = "311"
x = DNUM(i)
```

DO Statement

Statement: *Marks the beginning of a DO construct. The DO construct controls the repeated execution of a block of statements or constructs. (This repeated execution is called a loop.)*

Syntax

A DO construct takes one of the following forms:

Block Form:

```
[name:] DO [label[, ] ] [loop-control]
   block
[label] term-stmt
```

Nonblock Form:

```
DO label [,] [loop-control]
   block
[label] ex-term-stmt
```

name	(Optional) Is the name of the DO construct.
label	(Optional) Is a statement label identifying the terminal statement.

loop-control	Is a DO iteration (see Iteration Loop Control) or a DO WHILE statement.
block	Is a sequence of zero or more statements or constructs.
term-stmt	Is the terminal statement for the block form of the construct.
ex-term-stmt	Is the terminal statement for the nonblock form of the construct.

Description

The terminal statement (*term-stmt*) for a block DO construct is an END DO or CONTINUE statement. If the block DO statement contains a label, the terminal statement must be identified with the same label. If no label appears, the terminal statement must be an END DO statement.

If a construct name is specified in a block DO statement, the same name must appear in the terminal END DO statement. If no construct name is specified in the block DO statement, no name can appear in the terminal END DO statement.

The terminal statement (*ex-term-stmt*) for a nonblock DO construct is an executable statement (or construct) that is identified by the label specified in the nonblock DO statement. A nonblock DO construct can share a terminal statement with another nonblock DO construct. A block DO construct cannot share a terminal statement.

The following cannot be terminal statements for nonblock DO constructs:

- CONTINUE (allowed if it is a shared terminal statement)
- CYCLE
- END (for a program or subprogram)
- EXIT
- GO TO (unconditional or assigned)
- Arithmetic IF
- RETURN
- STOP

The nonblock DO construct is an obsolescent feature in Fortran 95 and Fortran 90.

Example

The following example shows a simple block DO construct (contains no iteration count or DO WHILE statement):

```
DO
  READ *, N
  IF (N == 0) STOP
  CALL SUBN
END DO
```

The DO block executes repeatedly until the value of zero is read. Then the DO construct terminates.

The following example shows a named block DO construct:

```
LOOP_1: DO I = 1, N
          A(I) = C * B(I)
        END DO LOOP_1
```

The following example shows a nonblock DO construct with a shared terminal statement:

```
    DO 20 I = 1, N
    DO 20 J = 1 + I, N
20 RESULT(I,J) = 1.0 / REAL(I + J)
```

The following two program fragments are also examples of DO statements:

```
C   Initialize the even elements of a 20-element real array
C
    DIMENSION array(20)
    DO j = 2, 20, 2
      array(j) = 12.0
    END DO
C
C   Perform a function 11 times
C
    DO k = -30, -60, -3
      int = j / 3
      isb = -9 - k
      array(isb) = MyFunc (int)
    END DO
```

The following shows the final value of a DO variable (in this case 11):

```
DO j = 1, 10
  WRITE (*, '(I5)') j
END DO
WRITE (*, '(I5)') j
```

See Also

- C to D
- CONTINUE
- CYCLE
- EXIT
- DO WHILE
- Execution Control
- DO Constructs

DO Directive

OpenMP* Fortran Compiler Directive: *Specifies that the iterations of the immediately following DO loop must be executed in parallel.*

Syntax

```
c$OMP DO [clause[[,] clause] ... ]
   do_loop
[c$OMP END DO [NOWAIT]]
```

c — Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

clause — Is one of the following:

- FIRSTPRIVATE (list)
- LASTPRIVATE (list)
- ORDERED

  Must be used if ordered sections are contained in the dynamic extent of the DO directive. For more information about ordered sections, see the ORDERED directive.
- PRIVATE (list)
- REDUCTION (operator | intrinsic : list)
- SCHEDULE (type[, chunk])

  Specifies how iterations of the DO loop are divided among the threads of the team. *chunk* must be a scalar integer expression. The following four *type*s are permitted, three of which allow the optional parameter *chunk*:

Type	Effect
STATIC	Divides iterations into contiguous pieces by dividing the number of iterations by the number of threads in the

Type	Effect
	team. Each piece is then dispatched to a thread before loop execution begins. If *chunk* is specified, iterations are divided into pieces of a size specified by *chunk*. The pieces are statically dispatched to threads in the team in a round-robin fashion in the order of the thread number.
DYNAMIC	Can be used to get a set of iterations dynamically. It defaults to 1 unless *chunk* is specified. If *chunk* is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically gets the next set of iterations.
GUIDED	Can be used to specify a minimum number of iterations. It defaults to 1 unless *chunk* is specified. If *chunk* is specified, the chunksize is reduced exponentially with each succeeding dispatch. The *chunk* specifies the minimum number of iterations to dispatch each time. If there

Type	Effect
	are less than *chunk* iterations remaining, the rest are dispatched.
AUTO[1]	Delegates the scheduling decision until compile time or run time. The schedule is processor dependent. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
RUNTIME[1]	Defers the scheduling decision until run time. You can choose a schedule type and chunksize at run time by using the environment variable OMP_SCHEDULE.

[1]No *chunk* is permitted for this type.

If the SCHEDULE clause is not used, the default schedule type is STATIC.

do_loop Is a DO iteration (an iterative DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer.
The iterations of the DO loop are distributed across the existing team of threads. The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.
You cannot branch out of a DO loop associated with a DO directive.

If used, the END DO directive must appear immediately after the end of the loop. If you do not specify an END DO directive, an END DO directive is assumed at the end of the DO loop.

If you specify NOWAIT in the END DO directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instruction following the loop without waiting for the other members of the team to finish the DO directive.

Parallel DO loop control variables are block-level entities within the DO loop. If the loop control variable also appears in the LASTPRIVATE list of the parallel DO, it is copied out to a variable of the same name in the enclosing PARALLEL region. The variable in the enclosing PARALLEL region must be SHARED if it is specified in the LASTPRIVATE list of a DO directive.

Only a single SCHEDULE clause and ORDERED clause can appear in a DO directive.

DO directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, the loop iteration variable is private by default, and it is not necessary to explicitly declare it. The END DO directive is optional:

```
c$OMP PARALLEL
c$OMP DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END DO
c$OMP END PARALLEL
```

If there are multiple independent loops within a parallel region, you can use the NOWAIT option to avoid the implied BARRIER at the end of the DO directive, as follows:

```
c$OMP PARALLEL
c$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END DO NOWAIT
c$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      END DO
c$OMP END DO NOWAIT
c$OMP END PARALLEL
```

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially, as follows:

```
c$OMP PARALLEL
c$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      END DO
c$OMP END PARALLEL
      CALL REVERSE(I)
```

In this case, the value of I at the end of the parallel region equals N+1, as in the sequential case.

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
c$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO
      ...
      SUBROUTINE WORK(K)
c$OMP ORDERED
      WRITE(*,*) K
c$OMP END ORDERED
```

See Also

- C to D
- OpenMP Fortran Compiler Directives
- Rules for General Directives that Affect DO Loops

DO WHILE

Statement: *Executes the range of a DO construct while a specified condition remains true.*

Syntax

DO [*label*[,]] WHILE (*expr*)

label	(Optional) Is a label specifying an executable statement in the same program unit.
expr	Is a scalar logical (test) expression enclosed in parentheses.

Description

Before each execution of the DO range, the logical expression is evaluated. If it is true, the statements in the body of the loop are executed. If it is false, the DO construct terminates and control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement.

You can transfer control out of a DO WHILE loop but not into a loop from elsewhere in the program.

Example

The following example shows a DO WHILE statement:

```
CHARACTER*132 LINE

...

I = 1

DO WHILE (LINE(I:I) .EQ. ' ')

  I = I + 1

END DO
```

The following examples show required and optional END DO statements:

```
Required            Optional

DO WHILE (I .GT. J)          DO 10 WHILE (I .GT. J)

  ARRAY(I,J) = 1.0             ARRAY(I,J) = 1.0

  I = I - 1                    I = I - 1

END DO                    10 END DO
```

The following shows another example:

```
CHARACTER(1) input

input = ' '

DO WHILE ((input .NE. 'n') .AND. (input .NE. 'y'))

   WRITE (*, '(A)') 'Enter y or n: '

   READ (*, '(A)') input

END DO
```

See Also

- C to D
- CONTINUE
- CYCLE
- EXIT
- DO

- Execution Control
- DO Constructs

DOT_PRODUCT

Transformational Intrinsic Function (Generic):
Performs dot-product multiplication of numeric or logical vectors (rank-one arrays).

Syntax

```
result = DOT_PRODUCT (vector_a,vector_b)
```

vector_a	(Input) Must be a rank-one array of numeric (integer, real, or complex) or logical type.
vector_b	(Input) Must be a rank-one array of numeric type if *vector_a* is of numeric type, or of logical type if *vector_a* is of logical type. It must be the same size as *vector_a*.

Results

The result is a scalar whose type depends on the types of *vector_a* and *vector_b*.

If *vector_a* is of type integer or real, the result value is SUM (*vector_a** *vector_b*).

If *vector_a* is of type complex, the result value is SUM (CONJG (*vector_a*)* *vector_b*).

If *vector_a* is of type logical, the result has the value ANY (*vector_a*.AND. *vector_b*).

If either rank-one array has size zero, the result is zero if the array is of numeric type, and false if the array is of logical type.

Example

DOT_PRODUCT ((/1, 2, 3/), (/3, 4, 5/)) has the value 26, calculated as follows:

```
((1 x 3) + (2 x 4) + (3 x 5)) = 26
```

DOT_PRODUCT ((/ (1.0, 2.0), (2.0, 3.0) /), (/ (1.0, 1.0), (1.0, 4.0) /)) has the value (17.0, 4.0).

DOT_PRODUCT ((/ .TRUE., .FALSE. /), (/ .FALSE., .TRUE. /)) has the value false.

The following shows another example:

```
  I = DOT_PRODUCT((/1,2,3/), (/4,5,6/)) ! returns
                                        !  the value 32
```

See Also

- C to D
- PRODUCT
- MATMUL
- TRANSPOSE

DOUBLE COMPLEX

Statement: *Specifies the DOUBLE COMPLEX data type.*

Syntax

```
DOUBLE COMPLEX
```

A COMPLEX(8) or DOUBLE COMPLEX constant is a pair of constants that represents a complex number. One of the pair must be a double-precision real constant, the other can be an integer, single-precision real, or double-precision real constant.

A COMPLEX(8) or DOUBLE COMPLEX constant occupies 16 bytes of memory and is interpreted as a complex number.

The rules for DOUBLE PRECISION (REAL(8)) constants also apply to the double precision portion of COMPLEX(KIND=8) or DOUBLE COMPLEX constants. (For more information, see REAL and DOUBLE PRECISION.)

The DOUBLE PRECISION constants in a COMPLEX(8) or DOUBLE COMPLEX constant have IEEE* T_floating format.

Example

```
DOUBLE COMPLEX vector, arrays(7,29)

DOUBLE COMPLEX pi, pi2 /3.141592654,6.283185308/
```

The following examples demonstrate valid and invalid COMPLEX(KIND=8) or DOUBLE COMPLEX constants:

Valid

(547.3E0_8,-1.44_8)

(1.7039E0,-1.7039D0)

(+12739D3,0.D0)

(1.23D0,)	Second constant missing.
(1D1,2H12)	Hollerith constants not allowed.
(1,1.2)	Neither constant is DOUBLE PRECISION; this is a valid single-precision real constant.

See Also

- C to D
- General Rules for Complex Constants
- COMPLEX Statement
- Complex Data Types
- DOUBLE PRECISION
- REAL

DOUBLE PRECISION

Statement: *Specifies the DOUBLE PRECISION data type.*

Syntax

```
DOUBLE PRECISION
```

A REAL(8) or DOUBLE PRECISION constant has more than twice the accuracy of a REAL(4) number, and greater range.

A REAL(8) or DOUBLE PRECISION constant occupies eight bytes of memory. The number of digits that precede the exponent is unlimited, but typically only the leftmost 15 digits are significant.

IEEE* T_floating format is used.

For more information, see General Rules for Real Constants.

Example

```
DOUBLE PRECISION varnam

DOUBLE PRECISION,PRIVATE :: zz
```

Table 789: Valid REAL(8) or DOUBLE PRECISION constants

123456789D+5

123456789E+5_8

+2.7843D00

-.522D-12

2E200_8

2.3_8

3.4E7_8

Table 790: Invalid REAL(8) or DOUBLE PRECISION constants

-.25D0_2	2 is not a valid kind type for reals.
+2.7182812846182	No D exponent designator is present; this is a valid single-precision constant.
123456789.D400	Too large for any double-precision format.
123456789.D-400	Too small for any double-precision format.

See Also

- C to D
- REAL Statement
- REAL(8) or DOUBLE PRECISION Constants
- Data Types, Constants, and Variables
- Real Data Types

DPROD

Elemental Intrinsic Function (Specific):
Produces a higher precision product. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax

```
result = DPROD (x, y)
```

x	(Input) Must be of type REAL(4) or REAL(8).
y	(Input) Must have the same type and kind parameters as *x*.

Results

If *x* and *y* are of type REAL(4), the result type is double-precision real (REAL(8) or REAL*8). If *x* and *y* are of type REAL(8), the result type is REAL(16). The result value is equal to *x** *y*.

The setting of compiler options specifying real size can affect this function.

Example

DPROD (2.0, -4.0) has the value -8.00D0.

DPROD (5.0D0, 3.0D0) has the value 15.00Q0.

The following shows another example:

```
REAL(4) e
REAL(8) d
e = 123456.7
d = 123456.7D0
! DPROD (e,e) returns 15241557546.4944
! DPROD (d,d) returns 15241556774.889999281387426890432 8
```

DRAND, DRANDM

Portability Functions: *Return double-precision random numbers in the range 0.0 through 1.0.*

Module

USE IFPORT

Syntax

```
result = DRAND (iflag)
result = DRANDM (iflag)
```

iflag	(Input) INTEGER(4). Controls the way the random number is selected.

Results

The result type is REAL(8). Return values are:

Value of *iflag*	Selection process
1	The generator is restarted and the first random value is selected.
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using *iflag*, then restarted, and the first random value is selected.

There is no difference between DRAND and DRANDM. Both functions are included to insure portability of existing code that references one or both of them.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
REAL(8) num
INTEGER(4) f
f=1
CALL print_rand
f=0
CALL print_rand
f=22
CALL print_rand
CONTAINS
  SUBROUTINE print_rand
    num = drand(f)
    print *, 'f= ',f,':',num
  END SUBROUTINE
END
```

See Also

- C to D
- RANDOM_NUMBER
- RANDOM_SEED

DRANSET

Portability Subroutine: *Sets the seed for the random number generator.*

Module

USE IFPORT

Syntax

CALL DRANSET (*seed*)

seed (Input) REAL(8). The reset value for the seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- C to D

RANGET

DREAL

Elemental Intrinsic Function (Specific):
Converts the real part of a double complex argument to double-precision type. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax

```
result = DREAL (a)
```

a (Input) Must be of type double complex (COMPLEX(8) or COMPLEX*16).

Results

The result type is double precision real (REAL(8) or REAL*8).

Example

DREAL ((2.0d0, 3.0d0)) has the value 2.0d0.

See Also

- C to D
- REAL

DSHIFTL

Elemental Intrinsic Function (Specific): *Selects the left 64 bits after shifting a 128-bit integer value to the left. This function cannot be passed as an actual argument.*

Syntax

`result = DSHIFTL (ileft,iright,ishift)`

ileft	(Input) INTEGER(8).
iright	(Input) INTEGER(8).
ishift	(Input) INTEGER(8). Must be nonnegative and less than or equal to 64. This is the shift count.

Results

The result type is INTEGER(8). The result value is the 64-bit value starting at bit 128 - *ishift* of the 128-bit concatenation of the values of *ileft* and *iright*.

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTL (ILEFT, IRIGHT, 16_8)  ! prints 1306643199093243919
```

DSHIFTR

Elemental Intrinsic Function (Specific): *Selects the left 64 bits after shifting a 128-bit integer value to the right. This function cannot be passed as an actual argument.*

Syntax

`result = DSHIFTR (ileft,iright,ishift)`

ileft	(Input) INTEGER(8).
iright	(Input) INTEGER(8).

ishift (Input) INTEGER(8). Must be nonnegative and less than or equal to 64. This is the shift count.

Results

The result type is INTEGER(8). The result value is the 64-bit value starting at bit 64 + *ishift* of the 128-bit concatenation of the values of *ileft* and *iright*.

Example

Consider the following:

```
INTEGER(8) ILEFT / Z'111122221111222' /
INTEGER(8) IRIGHT / Z'FFFFFFFFFFFFF' /
PRINT *, DSHIFTR (ILEFT, IRIGHT, 16_8)  ! prints 1306606910610341887
```

DTIME

Portability Function: *Returns the elapsed CPU time since the start of program execution when first called, and the elapsed execution time since the last call to DTIME thereafter.*

Module

USE IFPORT

Syntax

```
result = DTIME (tarray)
```

tarray (Output) REAL(4). A rank one array with two elements:

- *tarray*(1) - Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code.
- *tarray*(2) - Elapsed system time, which is time spent executing privileged code (code in the Windows Executive).

Results

The result type is REAL(4). The result is the total CPU time, which is the sum of *tarray*(1) and *tarray*(2). If an error occurs, -1 is returned.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
REAL(4) I, TA(2)
I = DTIME(TA)
write(*,*) 'Program has been running for', I, 'seconds.'
write(*,*) ' This includes', TA(1), 'seconds of user time and', &
& TA(2), 'seconds of system time.'
```

See Also

- C to D
- DATE_AND_TIME
- CPU_TIME

E to F

ELEMENTAL

Keyword: *Asserts that a user-defined procedure is a restricted form of pure procedure. This is a Fortran 95 feature.*

Description

To specify an elemental procedure, use this keyword in a FUNCTION or SUBROUTINE statement.

An explicit interface must be visible to the caller of an ELEMENTAL procedure.

An elemental procedure can be passed an array, which is acted upon one element at a time.

For functions, the result must be scalar; it cannot have the POINTER or ALLOCATABLE attribute.

Dummy arguments have the following restrictions:

- They must be scalar.
- They cannot have the POINTER or ALLOCATABLE attribute.

- They (or their subobjects) cannot appear in a specification expression except as an argument to one of the intrinsic functions BIT_SIZE, LEN, KIND, or the numeric inquiry functions.
- They cannot be *.
- They cannot be dummy procedures.

If the actual arguments are all scalar, the result is scalar. If the actual arguments are array valued, the values of the elements (if any) of the result are the same as if the function or subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

Elemental procedures are pure procedures and all rules that apply to pure procedures also apply to elemental procedures.

Example

Consider the following:

```
MIN (A, 0, B)             ! A and B are arrays of shape (S, T)
```

In this case, the elemental reference to the **MIN**intrinsic function is an array expression whose elements have the following values:

```
MIN (A(I,J), 0, B(I,J)), I = 1, 2, ..., S, J = 1, 2, ..., T
```

See Also

- E to F
- FUNCTION
- SUBROUTINE
- Determining When Procedures Require Explicit Interfaces
- Optional Arguments

ELLIPSE, ELLIPSE_W (W*32, W*64)

Graphics Functions: *Draw a circle or an ellipse using the current graphics color.*

Module

USE IFQWIN

Syntax

```
result = ELLIPSE (control,x1,y1,x2,y2)
```

```
result = ELLIPSE_W (control,wx1,wy1,wx2,wy2)
```

control	(Input) INTEGER(2). Fill flag. Can be one of the following symbolic constants: • $GFILLINTERIOR - Fills the figure using the current color and fill mask. • $GBORDER - Does not fill the figure.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the ellipse is clipped or partially out of bounds, the ellipse is considered successfully drawn, and the return is 1. If the ellipse is drawn completely out of bounds, the return is 0.

The border is drawn in the current color and line style.

When you use ELLIPSE, the center of the ellipse is the center of the bounding rectangle defined by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). When you use ELLIPSE_W, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The control option given by $GFILLINTERIOR is equivalent to a subsequent call to the FLOODFILLRGB function using the center of the ellipse as the start point and the current color (set by SETCOLORRGB) as the boundary color.

NOTE. The ELLIPSE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Ellipse routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Ellipse. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

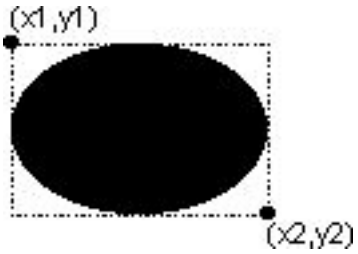
Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

This program draws the shape shown below.

```
! compile as QuickWin or Standard Graphics application
 USE IFQWIN
 INTEGER(2) dummy, x1, y1, x2, y2
 x1 = 80;  y1 = 50
 x2 = 240; y2 = 150
 dummy = ELLIPSE( $GFILLINTERIOR, x1, y1, x2, y2 )
 END
```



See Also

- E to F
- ARC
- FLOODFILLRGB
- GRSTATUS
- LINETO
- PIE
- POLYGON
- RECTANGLE
- SETCOLORRGB
- SETFILLMASK

ELSE Statement

See IF Construct.

ELSE Directive

See IF Directive Construct.

ELSEIF Directive

See IF Directive Construct.

ELSE IF

See IF Construct.

ELSE WHERE

Statement: *Marks the beginning of an ELSE WHERE block within a WHERE construct.*

Syntax

```
[name:]WHERE (mask-expr1)
   [where-body-stmt]...
[ELSE WHERE(mask-expr2) [name]
   [where-body-stmt]...]
[ ELSE WHERE[name]
   [where-body-stmt]...]
END WHERE [name]
```

name Is the name of the WHERE construct.

mask-expr1, mask-expr2 Are logical array expressions (called mask expressions).

where-body-stmt Is one of the following:

- An assignment statement of the form: array variable = array expression.

  The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.
- A WHERE statement or construct

Description

Every assignment statement following the ELSE WHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSE WHERE specifies "*mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSE WHERE statement.

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
ELSEWHERE
  raining = .TRUE.
END WHERE
```

The variables `temp`, `pressure`, and `raining` are all arrays.

See Also

- E to F
- WHERE

ENCODE

Statement: *Translates data from internal (binary) form to character form. It is comparable to using internal files in formatted sequential WRITE statements.*

Syntax

```
ENCODE (c,f,b[, IOSTAT=i-var] [, ERR=label]) [io-list]
```

c	Is a scalar integer expression. It is the number of characters to be translated to internal form.
f	Is a format identifier. An error occurs if more than one record is specified.
b	Is a scalar or array reference. If *b* is an array reference, its elements are processed in the order of subscript progression. *b* contains the characters to be translated to internal form.

i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs and as zero if no error occurs (see I/O Status Specifier).
label	Is the label of an executable statement that receives control if an error occurs.
io-list	Is an I/O list. An I/O list is either an implied-DO list or a simple list of variables (except for assumed-size arrays). The list contains the data to be translated to character form. The interaction between the format specifier and the I/O list is the same as for a formatted I/O statement.

The number of characters that the ENCODE statement can translate depends on the data type of *b*. For example, an INTEGER(2) array can contain two characters per element, so that the maximum number of characters is twice the number of elements in that array.

The maximum number of characters a character variable or character array element can contain is the length of the character variable or character array element.

The maximum number of characters a character array can contain is the length of each element multiplied by the number of elements.

Example

Consider the following:

```
     DIMENSION K(3)
     CHARACTER*12 A,B
     DATA A/'123456789012'/
     ENCODE(12,100,A) K
100  FORMAT(3I4)
     ENCODE(12,100,B) K(3), K(2), K(1)
```

The 12 characters are stored in array K:

```
K(1) = 1234
K(2) = 5678
K(3) = 9012
```

The ENCODE statement translates the values K(3), K(2), and K(1) to character form and stores the characters in the character variable B.:

```
B = '901256781234'
```

See Also

- E to F
- READ
- WRITE
- DECODE

END

Statement: *Marks the end of a program unit. It takes one of the following forms:*

Syntax

```
END [PROGRAM [program-name]]

END [FUNCTION [function-name]]

END [SUBROUTINE [subroutine-name]]

END [MODULE [module-name]]

END [BLOCK DATA [block-data-name]]
```

For internal procedures and module procedures, you must specify the FUNCTION and SUBROUTINE keywords in the END statement; otherwise, the keywords are optional.

In main programs, function subprograms, and subroutine subprograms, END statements are executable and can be branch target statements. If control reaches the END statement in these program units, the following occurs:

- In a main program, execution of the program terminates.
- In a function or subroutine subprogram, a RETURN statement is implicitly executed.

The END statement cannot be continued in a program unit, and no other statement in the program unit can have an initial line that appears to be the program unit END statement.

The END statements in a module or block data program unit are nonexecutable.

Example

```
C   An END statement must be the last statement in a program
C   unit:
    PROGRAM MyProg
    WRITE (*, '("Hello, world!")')
    END
C
C   An example of a named subroutine
C
    SUBROUTINE EXT1 (X,Y,Z)
      Real, Dimension (100,100) :: X, Y, Z
    END SUBROUTINE EXT1
```

See Also

- E to F
- Program Units and Procedures
- Branch Statements

END DO

Statement: *Marks the end of a DO or DO WHILE loop.*

Syntax

```
END DO
```

Description

There must be a matching END DO statement for every DO or DO WHILE statement that does not contain a label reference.

An END DO statement can terminate only one DO or DO WHILE statement. If you name the DO or DO WHILE statement, the END DO statement can specify the same name.

Example

The following examples both produce the same output:

```
  DO ivar = 1, 10
    PRINT ivar
  END DO
  ivar = 0
do2: DO WHILE (ivar .LT. 10)
        ivar = ivar + 1
        PRINT ivar
     END DO do2
```

See Also

- E to F
- DO
- DO WHILE
- CONTINUE

ENDIF Directive

See IF Directive Construct.

END IF

See IF Construct.

ENDFILE

Statement: *For sequential files, writes an end-of-file record to the file and positions the file after this record (the terminal point). For direct access files, truncates the file after the current record.*

Syntax

It can have either of the following forms:

```
ENDFILE ([UNIT=] io-unit[, ERR= label] [, IOSTAT=i-var])
ENDFILE io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

If the unit specified in the ENDFILE statement is not open, the default file is opened for unformatted output.

An end-of-file record can be written only to files with sequential organization that are accessed as formatted-sequential or unformatted-segmented sequential files. An ENDFILE performed on a direct access file always truncates the file.

End-of-file records should not be written in files that are read by programs written in a language other than Fortran.

NOTE. If you use compiler option vms and an ENDFILE is performed on a sequential unit, an actual one byte record containing a CTRL+Zis written to the file. If this option is not specified, an internal ENDFILE flag is set and the file is truncated. The option does not affect ENDFILE on relative files; such files are truncated.

If a parameter of the ENDFILE statement is an expression that calls a function, that function must not cause an I/O statement or the EOF intrinsic functionto be executed, because unpredictable results can occur.

Example

The following statement writes an end-of-file record to I/O unit 2:

```
ENDFILE 2
```

Suppose the following statement is specified:

```
ENDFILE (UNIT=9, IOSTAT=IOS, ERR=10)
```

An end-of-file record is written to the file connected to unit 9. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable `IOS`.

The following shows another example:

```
WRITE (6, *) x
ENDFILE 6
REWIND 6
READ (6, *) y
```

See Also

- E to F
- BACKSPACE
- REWIND
- Data Transfer I/O Statements
- Branch Specifiers

END FORALL

Statement: *Marks the end of a FORALL construct. See FORALL.*

END INTERFACE

Statement: *Marks the end of an INTERFACE block. See INTERFACE.*

MAP...END MAP

Statement: *Specifies mapped field declarations that are part of a UNION declaration within a STRUCTURE declaration. See STRUCTURE.*

Example

```
UNION
  MAP
     CHARACTER*20 string
  END MAP
  MAP
     INTEGER*2 number(10)
  END MAP
END UNION
UNION
  MAP
     RECORD /Cartesian/ xcoord, ycoord
  END MAP
  MAP
     RECORD /Polar/ length, angle
  END MAP
END UNION
```

SELECT CASE...END SELECT

Statement: *Transfers program control to a selected block of statements according to the value of a controlling expression. CASE.*

Example

```
CHARACTER*1 cmdchar

. . .

Files: SELECT CASE (cmdchar)

  CASE ('0')

    WRITE (*, *) "Must retrieve one to nine files"

  CASE ('1':'9')

    CALL RetrieveNumFiles (cmdchar)

  CASE ('A', 'a')

    CALL AddEntry

  CASE ('D', 'd')

    CALL DeleteEntry

  CASE ('H', 'h')

    CALL Help

  CASE DEFAULT

    WRITE (*, *) "Command not recognized; please re-enter"

END SELECT Files
```

STRUCTURE...END STRUCTURE

Statement: *Defines the field names, types of data within fields, and order and alignment of fields within a record structure. Fields and structures can be initialized, but records cannot be initialized.*

Syntax

```
STRUCTURE [/structure-name/] [field-namelist]
```

```
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END STRUCTURE
```

structure-name Is the name used to identify a structure, enclosed by slashes. Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names, but structures can share names with variables (scalar or array), record fields, PARAMETER constants, and common blocks. Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and is optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name.
Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structures within the calling and called subprograms must match in type, order, and dimension.

field-namelist Is a list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations.

field-declaration Also called the declaration body. A *field-declaration* consists of any combination of the following:

- Type declarations

  These are ordinary Fortran data type declarations.
- Substructure declarations

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.
- Union declarations

  A union declaration is composed of one or more mapped field declarations.
- PARAMETER statements

PARAMETER statements can appear in a structure declaration, but cannot be given a data type within the declaration block.

Type declarations for PARAMETER names must precede the PARAMETER statement and be outside of a STRUCTURE declaration, as follows:

```
   INTEGER*4 P
 STRUCTURE /ABC/
   PARAMETER (P=4)
   REAL*4 F
 END STRUCTURE
   REAL*4 A(P)
```

The Fortran 90 derived type replaces STRUCTURE and RECORD constructs, and should be used in writing new code. See Derived Data Types.

Unlike type declaration statements, structure declarations do not create variables. Structured variables (records) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type declaration statement. The difference is that aggregate items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important, because this ordering determines the order of the fields in records.

In a structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

By default, fields are aligned on natural boundaries; misaligned fields are padded as necessary. To avoid padding of records, you should lay out structures so that all fields are naturally aligned.

To pack fields on arbitrary byte boundaries, you must specify a compiler option. You can also specify alignment for fields by using the OPTIONS or PACK general directive.

A field name must not be the same as any intrinsic or user-defined operator (for example, EQ cannot be used as a field name).

Compatibility

An item can be a RECORD statement that references a previously defined structure type:

```
STRUCTURE /full_address/
  RECORD  /full_name/ personsname
  RECORD  /address/   ship_to
  INTEGER*1           age
  INTEGER*4           phone
END STRUCTURE
```

You can specify a particular item by listing the sequence of items required to reach it, separated by a period (.). Suppose you declare a structure variable, `shippingaddress`, using the `full_address`structure defined in the previous example:

```
RECORD /full_address/ shippingaddress
```

In this case, the `age`item would then be specified by `shippingaddress.age`, the first name of the receiver by `shippingaddress.personsname.first_name`, and so on.

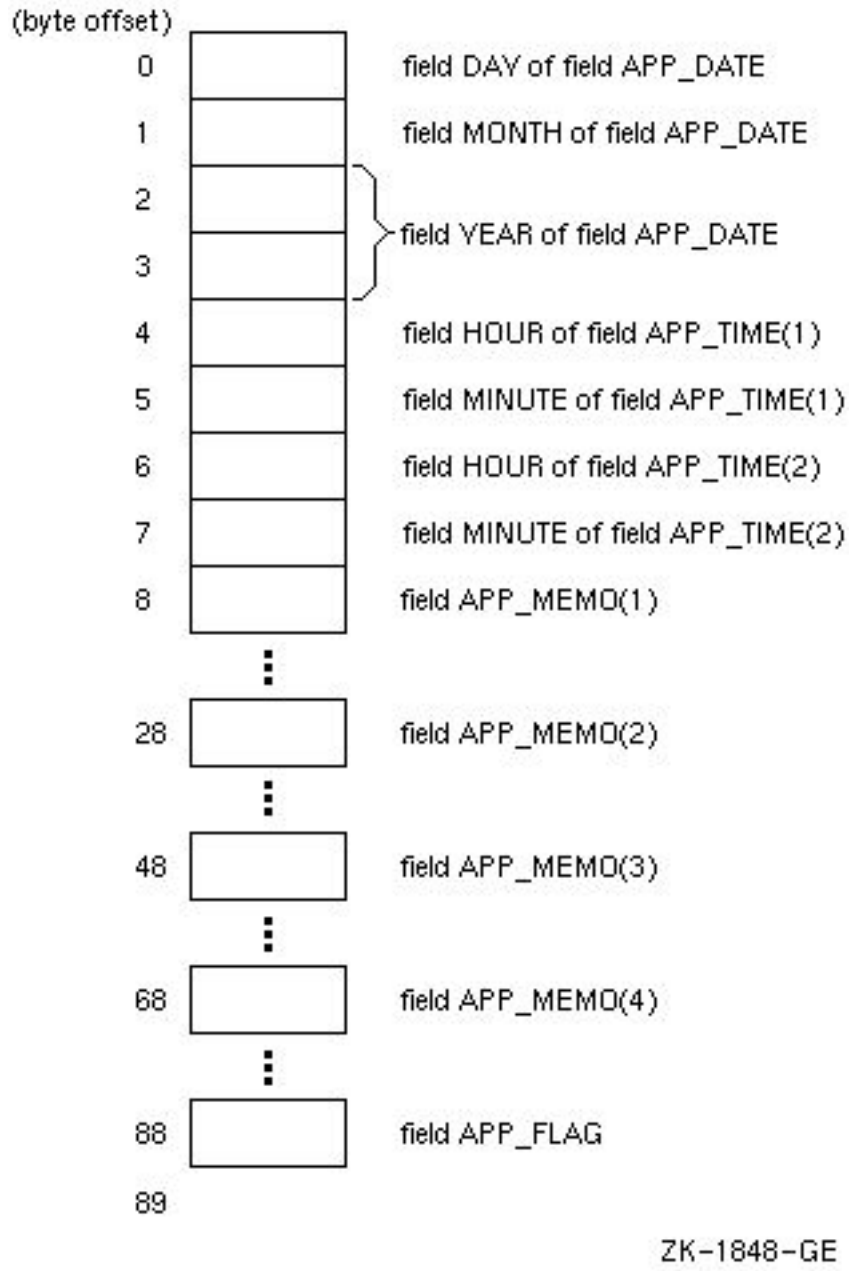
In the following example, the declaration defines a structure named `APPOINTMENT`. `APPOINTMENT`contains the structure `DATE`(field `APP_DATE`) as a substructure. It also contains a substructure named `TIME`(field `APP_TIME`, an array), a CHARACTER*20 array named `APP_MEMO`, and a LOGICAL*1 field named `APP_FLAG`.

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE
STRUCTURE /APPOINTMENT/
  RECORD /DATE/     APP_DATE
  STRUCTURE /TIME/  APP_TIME (2)
    INTEGER*1       HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20      APP_MEMO (4)
  LOGICAL*1         APP_FLAG
END STRUCTURE
```

The length of any instance of structure `APPOINTMENT` is 89 bytes.

The following figure shows the memory mapping of any record or record array element with the structure APPOINTMENT.

Figure 54: Memory Map of Structure APPOINTMENT


(byte offset)
0
1
2
3
4
5
6
7
8
28
48
68
88
89
field DAY of field APP_DATE
field MONTH of field APP_DATE
field YEAR of field APP_DATE
field HOUR of field APP_TIME(1)
field MINUTE of field APP_TIME(1)
field HOUR of field APP_TIME(2)
field MINUTE of field APP_TIME(2)
field APP_MEMO(1)
field APP_MEMO(2)
field APP_MEMO(3)
field APP_MEMO(4)
field APP_FLAG
ZK-1848-GE

See Also

- E to F
- S
- TYPE
- MAP...END MAP
- RECORD
- UNION...END UNION
- PACK Directive
- OPTIONS Directive
- Data Types, Constants, and Variables
- Record Structures

TYPE Statement (Derived Types)

Statement: *Declares a variable to be a derived type. It specifies the name of the user-defined type and the types of its components.*

Syntax

```
TYPE [[,type-attr-spec-list] :: ] name
component-definition
   [component-definition]. . .
END TYPE [ name ]
```

type-attr-spec-list	Is *access-spec* or BIND (C).
access-spec	Is the PUBLIC or PRIVATE keyword. The keyword can only be specified if the derived-type definition is in the specification part of a module.
name	Is the name of the derived data type. It must not be the same as the name of any intrinsic type, or the same as the name of a derived type that can be accessed from a module.
component-definition	Is one or more type declaration statements defining the component of derived type. The first component definition can be preceded by an optional PRIVATE or SEQUENCE statement. (Only one PRIVATE or SEQUENCE statement can appear in a given derived-type definition.)

If SEQUENCE is present, all derived types specified in component definitions must be sequence types.

A *component definition* takes the following form:

type[[, *attr*] ::] *component*[(*a-spec*)] [**char-len*] [*init-ex*]

type	Is a type specifier. It can be an intrinsic type or a previously defined derived type. (If the POINTER attribute follows this specifier, the type can also be any accessible derived type, including the type being defined.)
attr	Is an optional POINTER attribute for a pointer component, or an optional DIMENSION or ALLOCATABLE attribute for an array component. You cannot specify both the ALLOCATABLE and POINTER attribute. If DIMENSION is specified, it can be followed by an array specification. Each attribute can only appear once in a given *component-definition*.
component	Is the name of the component being defined.
a-spec	Is an optional array specification, enclosed in parentheses. If POINTER or ALLOCATABLE is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression. If the array bounds are not specified here, they must be specified following the DIMENSION attribute.
char-len	Is an optional scalar integer literal constant; it must be preceded by an asterisk (*). This parameter can only be specified if the component is of type CHARACTER.
init-ex	Is an initialization expression, or for pointer components, => NULL( ). This is a Fortran 95 feature.

If *init-ex* is specified, a double colon must appear in the component definition. The equals assignment symbol (=) can only be specified for nonpointer components.
The initialization expression is evaluated in the scoping unit of the type definition.

Description

If a name is specified following the END TYPE statement, it must be the same name that follows TYPE in the derived type statement.

A derived type can be defined only once in a scoping unit. If the same derived-type name appears in a derived-type definition in another scoping unit, it is treated independently.

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.

Two data entities have the same type if they are both declared to be of the same derived type (the derived-type definition can be accessed from a module or a host scoping unit).

If the entities are in different scoping units, they can also have the same derived type if they are declared with reference to different derived-type definitions, and if both derived-type definitions have all of the following:

- The same name
- A SEQUENCE statement (they both have sequence type)
- Components that agree in name, order, and attributes; components cannot be private

If BIND (C) is specified, the following rules apply:

- The derived type cannot be a SEQUENCE type.
- The derived type must have type parameters.
- Each component of the derived type must be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Example

```
!   DERIVED.F90
!   Define a derived-type structure,
!   type variables, and assign values
    TYPE member
      INTEGER age
      CHARACTER (LEN = 20) name
    END TYPE member
    TYPE (member) :: george
    TYPE (member) :: ernie
    george     = member( 33, 'George Brown' )
    ernie%age  = 56
    ernie%name = 'Ernie Brown'
    WRITE (*,*) george
    WRITE (*,*) ernie
    END
```

The following shows another example of a derived type:

```
TYPE mem_name
  SEQUENCE
  CHARACTER (LEN = 20) lastn
  CHARACTER (LEN = 20) firstn
  CHARACTER (len = 3) cos   ! this works because COS is a component name
END TYPE mem_name
TYPE member
    TYPE (mem_name) :: name
    SEQUENCE
    INTEGER age
    CHARACTER (LEN = 20) specialty
END TYPE member
```

In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
 TYPE (pair)
   INTEGER i, j
 END TYPE
 TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

The following example shows how you can use derived-type objects as components of other derived-type objects:

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
END TYPE
```

Objects of these derived types can then be used within a third derived-type specification, such as:

```
TYPE employee_data
  TYPE (employee_name) :: name
  TYPE (employee_addr) :: addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

See Also

- C to D
- E to F

- T to Z
- DIMENSION
- MAP...END MAP
- PRIVATE
- PUBLIC
- RECORD
- SEQUENCE
- STRUCTURE...END STRUCTURE
- Derived Data Types
- Default Initialization
- Structure Components
- Structure Constructors

Building Applications: Handling User-Defined Types

UNION...END UNION

Statements: *Define a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a structure declaration.*

Syntax

Each unique field or group of fields is defined by a separate map declaration.

```
UNION
   map-declaration
   map-declaration
   [map-declaration]
   . . .
   [map-declaration]
END UNION
```

map-declaration Takes the following form:

```
MAP
   field-declaration
```

```
    [field-declaration]
    . . .
    [field-declaration]
END MAP
```

field-declaration Is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a data field (having a data type) within a union. It can be of any intrinsic or derived type.

As with normal Fortran type declarations, data can be initialized in field declaration statements in union declarations. However, if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the fields declared within it.

Manipulating data by using union declarations is similar to using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are concurrently associated with a common storage location and the data residing there; with union declarations you can use one discrete storage location to alternately contain a variety of fields (arrays or variables).

With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

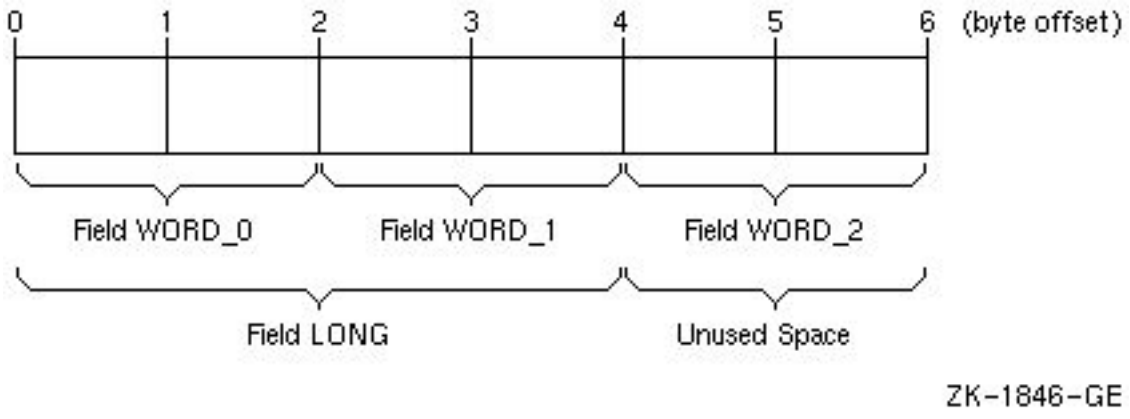
Example

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG:

```
STRUCTURE /WORDS_LONG/
  UNION
    MAP
      INTEGER*2  WORD_0, WORD_1, WORD_2
    END MAP
    MAP
      INTEGER*4  LONG
    END MAP
  END UNION
END STRUCTURE
```

The length of any record with the structure WORDS_LONG is 6 bytes. The following figure shows the memory mapping of any record with the structure WORDS_LONG:

Figure 55: Memory Map of Structure WORDS_LONG



In the following example, note how the first 40 characters in the string2 array are overlayed on 4-byte integers, while the remaining 20 are overlayed on 2-byte integers:

```
UNION
   MAP
     CHARACTER*20 string1, CHARACTER*10 string2(6)
   END MAP
   MAP
     INTEGER*2 number(10), INTEGER*4 var(10), INTEGER*2
+     datum(10)
   END MAP
END UNION
```

See Also

- E to F
- T to Z
- STRUCTURE...END STRUCTURE
- Record Structures

END WHERE

Statement: *Marks the end of a WHERE block. See WHERE.*

Example

```
WHERE (pressure <= 1.0)
  pressure = pressure + inc_pressure
  temp = temp - 5.0
ELSEWHERE
  raining = .TRUE.
END WHERE
```

Note that the variables `temp`, `pressure`, and `raining` are all arrays.

ENTRY

Statement: *Provides one or more entry points within a subprogram. It is not executable and must precede any CONTAINS statement (if any) within the subprogram.*

Syntax

```
ENTRY name[ ( [d-arg[,d-arg]...] ) [RESULT (r-name)] ]
```

name	Is the name of an entry point. If RESULT is specified, this entry name must not appear in any specification statement in the scoping unit of the function subprogram.
d-arg	(Optional) Is a dummy argument. The dummy argument can be an alternate return indicator (*) if the ENTRY statement is within a subroutine subprogram.
r-name	(Optional) Is the name of a function result. This name must not be the same as the name of the entry point, or the name of any other function or function result. This parameter can only be specified for function subprograms.

Description

ENTRY statements can only appear in external procedures or module procedures.

An ENTRY statement must not appear in a CASE, DO, IF, FORALL, or WHERE construct, or a nonblock DO loop.

When the ENTRY statement appears in a subroutine subprogram, it is referenced by a CALL statement. When the ENTRY statement appears in a function subprogram, it is referenced by a function reference.

An entry name within a function subprogram can appear in a type declaration statement.

Within the subprogram containing the ENTRY statement, the entry name must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement, and it must not appear in an EXTERNAL or INTRINSIC statement. For example, neither of the following are valid:

```
(1)  SUBROUTINE SUB(E)
     ENTRY E
     ...
(2)  SUBROUTINE SUB
     EXTERNAL E
     ENTRY E
     ...
```

The procedure defined by an ENTRY statement can reference itself if the function or subroutine was defined as RECURSIVE.

Dummy arguments can be used in ENTRY statements even if they differ in order, number, type and kind parameters, and name from the dummy arguments used in the FUNCTION, SUBROUTINE, and other ENTRY statements in the same subprogram. However, each reference to a function, subroutine, or entry must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement.

Dummy arguments can be referred to only in executable statements that follow the first SUBROUTINE, FUNCTION, or ENTRY statement in which the dummy argument is specified. If a dummy argument is not currently associated with an actual argument, the dummy argument is undefined and cannot be referenced. Arguments do not retain their association from one reference of a subprogram to another.

Example

```
C  This fragment writes a message indicating
C  whether num is positive or negative
   IF (num .GE. 0) THEN
     CALL Sign
   ELSE
     CALL Negative
   END IF
   ...
   END
SUBROUTINE Sign
   WRITE (*, *) 'It''s positive.'
   RETURN
   ENTRY Negative
   WRITE (*, *) 'It''s negative.'
   RETURN
END SUBROUTINE
```

See Also

- E to F
- Program Units and Procedures
- ENTRY Statements in Function Subprograms
- ENTRY Statements in Subroutine Subprograms

EOF

Inquiry Intrinsic Function (Generic): *Checks whether a file is at or beyond the end-of-file record.*

Syntax

```
result = EOF (unit)
```

unit	(Input) Must be of type integer. It represents a unit specifier corresponding to an open file. It cannot be zero unless you have reconnected unit zero to a unit other than the screen or keyboard.

Results

The result type is default logical. The value of the result is .TRUE. if the file connected to *unit* is at or beyond the end-of-file record; otherwise, .FALSE..

Example

```
!  Creates a file of random numbers, reads them back
       REAL x, total
       INTEGER count
       OPEN (1, FILE = 'TEST.DAT')
       DO I = 1, 20
         CALL RANDOM_NUMBER(x)
         WRITE (1, '(F6.3)') x * 100.0
       END DO
       CLOSE(1)
       OPEN (1, FILE = 'TEST.DAT')
       DO WHILE (.NOT. EOF(1))
         count = count + 1
         READ (1, *) value
         total = total + value
       END DO
100    IF ( count .GT. 0) THEN
         WRITE (*,*) 'Average is: ', total / count
       ELSE
         WRITE (*,*) 'Input file is empty '
       END IF
        STOP
       END
```

See Also

- E to F
- ENDFILE
- BACKSPACE
- REWIND

EOSHIFT

Transformational Intrinsic Function (Generic): *Performs an end-off shift on a rank-one array, or performs end-off shifts on all the complete rank-one sections along a given dimension of an array of rank two or greater. Elements are shifted off at one end of a section and copies of a boundary value are filled in at the other end. Different sections can have different boundary values and can be shifted by different amounts and in different directions.*

Syntax

```
result = EOSHIFT (array,shift [,boundary][,dim])
```

array (Input) Must be an array (of any data type).

shift (Input) Must be a scalar integer or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.

boundary (Input; optional) Must have the same type and kind parameters as *array*. It must be a scalar or an array with a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n). The *boundary* specifies a value to replace spaces left by the shifting procedure.
If *boundary* is not specified, it is assumed to have the following default values (depending on the data type of *array*):

array **Type**	*boundary* **Value**
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character(*len*)	*len* blanks

dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. If *dim* is omitted, it is assumed to be 1.

Results

The result is an array with the same type and kind parameters, and shape as *array*.

If *array* has rank one, the same shift is applied to each element. If an element is shifted off one end of the array, the *boundary* value is placed at the other end the array.

If *array* has rank greater than one, each section (s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) of the result is shifted as follows:

- By the value of *shift*, if *shift* is scalar
- According to the corresponding value in *shift*(s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n), if *shift* is an array

If an element is shifted off one end of a section, the *boundary* value is placed at the other end of the section.

The value of *shift* determines the amount and direction of the end- off shift. A positive *shift* value causes a shift to the left (in rows) or up (in columns). A negative *shift* value causes a shift to the right (in rows) or down (in columns).

Example

V is the array (1, 2, 3, 4, 5, 6).

EOSHIFT (V, SHIFT=2) shifts the elements in V to the *left* by 2 positions, producing the value (3, 4, 5, 6, 0, 0). 1 and 2 are shifted off the beginning and two elements with the default BOUNDARY value are placed at the end.

EOSHIFT (V, SHIFT= -3, BOUNDARY= 99) shifts the elements in V to the *right* by 3 positions, producing the value (99, 99, 99, 1, 2, 3). 4, 5, and 6 are shifted off the end and three elements with BOUNDARY value 99 are placed at the beginning.

M is the character array

```
[ 1  2  3 ]
[ 4  5  6 ]
[ 7  8  9 ].
```

EOSHIFT (M, SHIFT = 1, BOUNDARY = '*', DIM = 2) produces the result

```
[ 2  3  * ]
[ 5  6  * ]
[ 8  9  * ].
```

Each element in rows 1, 2, and 3 is shifted to the *left* by 1 position. This causes the first element in each row to be shifted off the beginning, and the BOUNDARY value to be placed at the end.

EOSHIFT (M, SHIFT = -1, DIM = 1) produces the result

```
[ 0  0  0 ]
[ 1  2  3 ]
[ 4  5  6 ].
```

Each element in columns 1, 2, and 3 is shifted *down* by 1 position. This causes the last element in each column to be shifted off the end and the BOUNDARY value to be placed at the beginning.

EOSHIFT (M, SHIFT = (/1, -1, 0/), BOUNDARY = (/ '*', '?', '/' /), DIM = 2) produces the result

```
[ 2  3  * ]
[ ?  4  5 ]
[ 7  8  9 ].
```

Each element in row 1 is shifted to the *left* by 1 position, causing the first element to be shifted off the beginning and the BOUNDARY value * to be placed at the end. Each element in row 2 is shifted to the *right* by 1 position, causing the last element to be shifted off the end and the BOUNDARY value ? to be placed at the beginning. No element in row 3 is shifted at all, so the specified BOUNDARY value is not used.

The following shows another example:

```
INTEGER shift(3)

CHARACTER(1) array(3, 3), AR1(3, 3)

array = RESHAPE ((/'A', 'D', 'G', 'B', 'E', 'H', &

                  'C', 'F', 'I'/), (/3,3/))

!        array is A B C

!                 D E F

!                 G H I

shift = (/-1, 1, 0/)

AR1 = EOSHIFT (array, shift, BOUNDARY = (/'*','?','#'/), DIM= 2)

! returns         * A B

!                 E F ?

!                 G H I
```

See Also

- E to F
- CSHIFT
- ISHFT
- ISHFTC
- TRANSPOSE

EPSILON

Inquiry Intrinsic Function (Generic): *Returns a positive model number that is almost negligible compared to unity in the model representing real numbers.*

Syntax

```
result = EPSILON (x)
```

x	(Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar of the same type and kind parameters as *x*. The result has the value b $^{1-p}$. Parameters *b* and *p* are defined in Model for Real Data.

EPSILON makes it easy to select a *delta* for algorithms (such as root locators) that search until the calculation is within *delta* of an estimate. If *delta* is too small (smaller than the decimal resolution of the data type), the algorithm might never halt. By scaling the value returned by EPSILON to the estimate, you obtain a *delta* that ensures search termination.

Example

If *x* is of type REAL(4), EPSILON (X) has the value 2 $^{-23}$.

See Also

- E to F
- PRECISION
- TINY
- Data Representation Models

EQUIVALENCE

Statement: *Specifies that a storage area is shared by two or more objects in a program unit. This causes total or partial storage association of the objects that share the storage area.*

Syntax

```
EQUIVALENCE (equiv-list) [,(equiv-list)]...
```

equiv-list	Is a list of two or more variable names, array elements, or substrings, separated by commas (also called an equivalence set). If an object of derived type is specified, it must be a sequence type. Objects cannot have the TARGET attribute. Each expression in a subscript or a substring reference must be an integer initialization expression. A substring must not have a length of zero.

Description

The following objects cannot be specified in EQUIVALENCE statements:

- A dummy argument

- An allocatable variable
- An automatic object
- A pointer
- An object of nonsequence derived type
- A derived-type object that has an allocatable or pointer component at any level
- A component of a derived-type object
- A function, entry, or result name
- A named constant
- A structure component
- A subobject of any of the above objects
- An object with either the DLLIMPORT or DLLEXPORT attribute
- A variable with the BIND attribute
- A variable in a common block that has the BIND attribute

The EQUIVALENCE statement causes all of the entities in one parenthesized list to be allocated storage beginning at the same storage location.

If an equivalence object has the PROTECTED attribute, all of the objects in the equivalence set must have the PROTECTED attribute.

Association of objects depends on their types, as follows:

Type of Object	**Type of Associated Object**
Intrinsic numeric [1]or numeric sequence	Can be of any of these types
Default character or character sequence	Can be of either of these types [2]
Any other intrinsic type	Must have the same type and kind parameters
Any other sequence type	Must have the same type

[1]Default integer, default real, double precision real, default complex, double complex, or default logical.

[2]The lengths do not have to be equal.

So, objects can be associated if they are of different numeric type. For example, the following is valid:

```
INTEGER A(20)
REAL Y(20)
EQUIVALENCE(A, Y)
```

Objects of default character do not need to have the same length. The following example associates character variable D with the last 4 (of the 6) characters of character array F:

```
CHARACTER(LEN=4) D
CHARACTER(LEN=3) F(2)
EQUIVALENCE(D, F(1)(3:))
```

Entities having different data types can be associated because multiple components of one data type can share storage with a single component of a higher-ranked data type. For example, if you make an integer variable equivalent to a complex variable, the integer variable shares storage with the real part of the complex variable.

The same storage unit cannot occur more than once in a storage sequence, and consecutive storage units cannot be specified in a way that would make them nonconsecutive.

Intel® Fortran lets you associate character and noncharacter entities, for example:

```
CHARACTER*1 char1(10)
REAL reala, realb
EQUIVALENCE (reala, char1(1))
EQUIVALENCE (realb, char1(2))
```

EQUIVALENCE statements require only the first subscript of a multidimensional array (unless the STRICT compiler directive is in effect). For example, the array declaration var(3,3), var(4) could appear in an EQUIVALENCE statement. The reference is to the fourth element of the array (var(1,2)), not to the beginning of the fourth row or column.

If you use the STRICT directive, the following rules apply to the kinds of variables and arrays that you can associate:

- If an EQUIVALENCE object is default integer, default real, double-precision real, default complex, default logical, or a sequenced derived type of all numeric or logical components, all objects in the EQUIVALENCE statement must be one of these types, though it is not necessary that they be the same type.

- If an EQUIVALENCE object is default character or a sequenced derived type of all character components, all objects in the EQUIVALENCE statement must be one of these types. The lengths do not need to be the same.
- If an EQUIVALENCE object is a sequenced derived type that is not purely numeric or purely character, all objects in the EQUIVALENCE statement must be the same derived type.
- If an EQUIVALENCE object is an intrinsic type other than the default (for example, INTEGER(1)), all objects in the EQUIVALENCE statement must be the same type and kind.

Example

The following EQUIVALENCE statement is invalid because it specifies the same storage unit for X(1) and X(2):

```
REAL, DIMENSION(2) :: X

REAL :: Y

EQUIVALENCE(X(1), Y), (X(2), Y)
```

The following EQUIVALENCE statement is invalid because A(1) and A(2) will not be consecutive:

```
REAL A(2)

DOUBLE PRECISION D(2)

EQUIVALENCE(A(1), D(1)), (A(2), D(2))
```

In the following example, the EQUIVALENCE statement causes the four elements of the integer array IARR to share the same storage as that of the double-precision variable DVAR:

```
DOUBLE PRECISION DVAR

INTEGER(KIND=2) IARR(4)

EQUIVALENCE(DVAR, IARR(1))
```

In the following example, the EQUIVALENCE statement causes the first character of the character variables KEY and STAR to share the same storage location. The character variable STAR is equivalent to the substring KEY(1:10).

```
CHARACTER KEY*16, STAR*10

EQUIVALENCE(KEY, STAR)
```

The following shows another example:

```
CHARACTER name, first, middle, last
DIMENSION name(60), first(20), middle(20), last(20)
EQUIVALENCE (name(1), first(1)), (name(21), middle(1))
EQUIVALENCE (name(41), last(1))
```

Consider the following:

```
CHARACTER (LEN = 4) :: a, b
CHARACTER (LEN = 3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

This causes the following alignment:

```
  1          2          3          4          5          6          7
a(1:1)     a(2:2)     a(3:3)     a(4:4)
                                 b(1:1)     b(2:2)     b(3:3)     b(4:4)
c(1)(1:1)  c(1)(2:2)  c(1)(3:3)  c(2)(1:1)  c(2)(2:2)  c(2)(3:3)
```

Note that the fourth element of a, the first element of b, and the first element of c(2) share the same storage unit.

See Also

- E to F
- EQUIVALENCE Statement
- Initialization Expressions
- Derived Data Types
- Storage Association
- STRICT Directive

ERF

Elemental Intrinsic Function (Generic):
Returns the error function of an argument.

Syntax

```
result = ERF (x)
```

x	(Input) Must be of type real.

Results

The result type is the same as *x*. The result is in the range -1 to 1.

ERF returns the error function of *x* defined as follows:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$$

Specific Name	Argument Type	Result Type
ERF	REAL(4)	REAL(4)
DERF	REAL(8)	REAL(8)
QERF	REAL(16)	REAL(16)

Example

ERF (1.0) has the value 0.842700794.

See Also

- E to F
- ERFC

ERFC

Elemental Intrinsic Function (Generic):
Returns the complementary error function of an argument.

Syntax

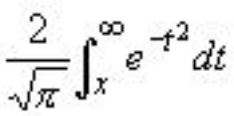
```
result = ERFC (x)
```

x (Input) Must be of type real.

Results

The result type is the same as *x*. The result is in the range 0 to 2.

ERFC returns 1 - ERF(x) and is defined as follows:



ERFC is provided because of the extreme loss of relative accuracy if ERF(x) is called for large *x* and the result is subtracted from 1.

Specific Name	Argument Type	Result Type
ERFC	REAL(4)	REAL(4)
DERFC	REAL(8)	REAL(8)
QERFC	REAL(16)	REAL(16)

Example

ERFC (1.0) has the value 0.1572992057.

See Also

- E to F
- ERF

ERRSNS

Intrinsic Subroutine (Generic): *Returns information about the most recently detected I/O system error condition. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL ERRSNS ([io_err] [,sys_err] [,stat] [,unit] [,cond])
```

io_err (Output; Optional) Is an integer variable or array element that stores the most recent Run-Time Library error number that occurred during program execution. (For a listing of error numbers, see *Building Applications*.)
A zero indicates no error has occurred since the last call to ERRSNS or since the start of program execution.

sys_err	(Output; Optional) Is an integer variable or array element that stores the most recent system error number associated with *io_err*. This code is one of the following: • On Windows* systems, it is the value returned by GETLASTERROR( ) at the time of the error. • On Linux* and Mac OS* X systems, it is an `errno` value. (See `errno(2)`.)
stat	(Output; Optional) Is an integer variable or array element that stores a status value that occurred during program execution. This value is always set to zero.
unit	(Output; Optional) Is an integer variable or array element that stores the logical unit number, if the last error was an I/O error.
cond	(Output; Optional) Is an integer variable or array element that stores the actual processor value. This value is always set to zero.

If you specify INTEGER(2) arguments, only the low-order 16 bits of information are returned or adjacent data can be overwritten. Because of this, it is best to use INTEGER(4) arguments.

The saved error information is set to zero after each call to ERRSNS.

Example

Any of the arguments can be omitted. For example, the following is valid:

```
CALL ERRSNS (SYS_ERR=I1, STAT=I2, UNIT=I4)
```

ETIME

Portability Function: *On single processor systems, returns the elapsed CPU time, in seconds, of the process that calls it. On multi-core or multi-processor systems, returns the elapsed wall-clock time, in seconds.*

Module

USE IFPORT

Syntax

```
result = ETIME (array)
```

array	(Output) REAL(4). Must be a rank one array with two elements: • *array(1)* – Elapsed user time, which is time spent executing user code. This value includes time running protected Windows subsystem code. On single processors, ETIME returns the elapsed CPU time, in seconds, of the process that calls it. On multiple processors, ETIME returns the elapsed wall-clock time, in seconds. • *array(2)* – Elapsed system time, which is time spent executing privileged code (code in the Windows Executive) on single processors; on multiple processors, this value is zero.

Results

The result type is REAL(4). The result is the total CPU time, which is the sum of *array(1)* and *array(2)*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
 USE IFPORT
 REAL(4) I, TA(2)
 I = ETIME(TA)
 write(*,*) 'Program has used', I, 'seconds of CPU time.'
 write(*,*) '  This includes', TA(1), 'seconds of user time and',  &
& TA(2), 'seconds of system time.'
```

See Also

- E to F
- DATE_AND_TIME

EXIT Statement

Statement: *Terminates execution of a DO construct.*

Syntax

```
EXIT [name]
```

name	(Optional) Is the name of the DO construct.

Description

The EXIT statement causes execution of the named (or innermost) DO construct to be terminated.

If a DO construct name is specified, the EXIT statement must be within the range of that construct.

Any DO variable present retains its last defined value.

An EXIT statement can be labeled, but it cannot be used to terminate a DO construct.

Example

The following example shows an EXIT statement:

```
LOOP_A : DO I = 1, 15
  N = N + 1
  IF (N > I) EXIT LOOP_A
END DO LOOP_A
```

The following shows another example:

```
CC  See CYCLE.F90 in the TBD for an example of EXIT in nested
CC  DO loops
CC  Loop terminates early if one of the data points is zero:
CC
    INTEGER numpoints, point
    REAL datarray(1000), sum
    sum = 0.0
    DO point = 1, 1000
      sum = sum + datarray(point)
      IF (datarray(point+1) .EQ. 0.0) EXIT
    END DO
```

See Also

- E to F
- DO
- DO WHILE

EXIT Subroutine

Intrinsic Subroutine (Generic): *Terminates program execution, closes all files, and returns control to the operating system. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL EXIT [( [status] )]
```

status	(Output; optional) Is an integer argument you can use to specify the image exit-status value.

The exit-status value may not be accessible after program termination in some application environments.

Example

```
    INTEGER(4) exvalue
! all is well, exit with 1
    exvalue = 1
    CALL EXIT(exvalue)
! all is not well, exit with diagnostic -4
    exvalue = -4
    CALL EXIT(exvalue)
! give no diagnostic, just exit
    CALL EXIT ( )
```

See Also

- E to F
- END
- ABORT

EXP

Elemental Intrinsic Function (Generic):
Computes an exponential value.

Syntax

```
result = EXP (x)
```

x	(Input) Must be of type real or complex.

Results

The result type is the same as *x*. The value of the result is e^{x}. If *x* is of type complex, its imaginary part is regarded as a value in radians.

Specific Name	Argument Type	Result Type
EXP	REAL(4)	REAL(4)
DEXP	REAL(8)	REAL(8)

Specific Name	Argument Type	Result Type
QEXP	REAL(16)	REAL(16)
CEXP [1]	COMPLEX(4)	COMPLEX(4)
CDEXP[2]	COMPLEX(8)	COMPLEX(8)
CQEXP	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CEXP.

[2]This function can also be specified as ZEXP.

Example

EXP (2.0) has the value 7.389056.

EXP (1.3) has the value 3.669297.

The following shows another example:

```
!  Given initial size and growth rate,
!  calculates the size of a colony at a given time.
      REAL sizei, sizef, time, rate
      sizei = 10000.0
      time = 40.5
      rate = 0.0875
      sizef = sizei * EXP (rate * time)
      WRITE (*, 100) sizef
100   FORMAT (' The final size is ', E12.6)
      END
```

See Also

- E to F
- LOG

EXPONENT

Elemental Intrinsic Function (Generic):
Returns the exponent part of the argument when represented as a model number.

Syntax

```
result = EXPONENT (x)
```

x	(Input) must be of type real.

Results

The result type is default integer. If *x* is not equal to zero, the result value is the exponent part of *x*. The exponent must be within default integer range; otherwise, the result is undefined.

If *x* is zero, the exponent of *x* is zero. For more information on the exponent part (*e*) in the real model, see Model for Real Data.

Example

EXPONENT (2.0) has the value 2.

If 4.1 is a REAL(4) value, EXPONENT (4.1) has the value 3.

The following shows another example:

```
REAL(4) r1, r2
REAL(8) r3, r4
r1 = 1.0
r2 = 123456.7
r3 = 1.0D0
r4 = 123456789123456.7
write(*,*) EXPONENT(r1) ! prints 1
write(*,*) EXPONENT(r2) ! prints 17
write(*,*) EXPONENT(r3) ! prints 1
write(*,*) EXPONENT(r4) ! prints 47
END
```

See Also

- E to F
- DIGITS
- RADIX
- FRACTION
- MAXEXPONENT
- MINEXPONENT
- Data Representation Models

EXTERNAL

Statement and Attribute: *Allows an external or dummy procedure to be used as an actual argument. (To specify intrinsic procedures as actual arguments, use the INTRINSIC attribute.)*

Syntax

The EXTERNAL attribute can be specified in a type declaration statement or an EXTERNAL statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] EXTERNAL [, *att-ls*] :: *ex-pro*[, *ex-pro*]...

Statement:

EXTERNAL [::]*ex-pro*[, *ex-pro*]...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ex-pro	Is the name of an external (user-supplied) procedure or dummy procedure.

Description

In a type declaration statement, only *functions* can be declared EXTERNAL. However, you can use the EXTERNAL *statement* to declare subroutines and block data program units, as well as functions, to be external.

The name declared EXTERNAL is assumed to be the name of an external procedure, even if the name is the same as that of an intrinsic procedure. For example, if SIN is declared with the EXTERNAL attribute, all subsequent references to SIN are to a user-supplied function named SIN, not to the intrinsic function of the same name.

You can include the name of a block data program unit in the EXTERNAL statement to force a search of the object module libraries for the block data program unit at link time. However, the name of the block data program unit must not be used in a type declaration statement.

If you want to describe a routine with greater detail, use the INTERFACE statement. This statement automatically declares a routine as EXTERNAL, and provides information on result types and argument types.

Example

The following example shows type declaration statements specifying the EXTERNAL attribute:

```
PROGRAM TEST

...

INTEGER, EXTERNAL :: BETA

LOGICAL, EXTERNAL :: COS

...

CALL SUB(BETA)       ! External function BETA is an actual argument
```

You can use a name specified in an EXTERNAL statement as an actual argument to a subprogram, and the subprogram can then use the corresponding dummy argument in a function reference or a CALL statement; for example:

```
EXTERNAL FACET

CALL BAR(FACET)

SUBROUTINE BAR(F)

EXTERNAL F

CALL F(2)
```

Used as an argument, a complete function reference represents a value, not a subprogram; for example, FUNC(B) represents a value in the following statement:

```
CALL SUBR(A, FUNC(B), C)
```

The following shows another example:

```
EXTERNAL MyFunc, MySub
C   MyFunc and MySub are arguments to Calc
    CALL Calc (MyFunc, MySub)
C   Example of a user-defined function replacing an
C   intrinsic
    EXTERNAL SIN
    x = SIN (a, 4.2, 37)
```

See Also

- E to F
- INTRINSIC
- Program Units and Procedures
- Type Declarations
- INTRINSIC
- Compatible attributes
- FORTRAN 66 Interpretation of the External Statement

FDATE

Portability Function and Subroutine: *Returns the current date and time as an ASCII string.*

Module

USE IFPORT

Syntax

Function Syntax

```
result =FDATE()
```

Subroutine Syntax:

```
CALL FDATE ( [string] )
```

string	(Output; optional) Character*(*). It is returned as a 24-character string in the form: `Mon Jan 31 04:37:23 2001`

Any value in *string* before the call is destroyed.

Results

The result of the function FDATE and the value of *string* returned by the subroutine FDATE(*string*) are identical. Newline and NULL are not included in the string.

When you use FDATE as a function, declare it as:

```
CHARACTER*24 FDATE
```

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

CHARACTER*24 today

!

CALL FDATE(today)

write (*,*), 'Today is ', today

!

write (*,*), 'Today is ', fdate()
```

See Also

- E to F
- DATE_AND_TIME

FGETC

Portability Function: *Reads the next available character from a file specified by a Fortran unit number.*

Module

USE IFPORT

Syntax

```
result = FGETC (lunit,char)
```

lunit	(Input) INTEGER(4). Unit number of a file. Must be currently connected to a file when the function is called.
char	(Output) CHARACTER*1. Next available character in the file. If *lunit* is connected to a console device, then no characters are returned until the Enter key is pressed.

Results

The result type is INTEGER(4). The result is zero if the read is successful, or -1 if an end-of-file is detected. A positive value is either a system error code or a Fortran I/O error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number).

If you use WRITE, READ, or any other Fortran I/O statements with *lunit*, be sure to read *Building Applications*: Portability Routines: Input and Output Routines.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

CHARACTER inchar

INTEGER istatus

istatus = FGETC(5,inchar)

PRINT *, inchar

END
```

See Also

- E to F
- GETCHARQQ
- READ

FIND

Statement: *Positions a direct access file at a particular record and sets the associated variable of the file to that record number. It is comparable*

to a direct access READ statement with no I/O list, and it can open an existing file. No data transfer takes place.

Syntax

```
FIND ([UNIT=] io-unit, REC= r[, ERR= label] [, IOSTAT= i-var])

FIND (io-unit 'r [, ERR=label] [, IOSTAT=i-var])
```

io-unit	Is a logical unit number. It must refer to a relative organization file (see Unit Specifier).
r	Is the direct access record number. It cannot be less than one or greater than the number of records defined for the file (see Record Specifier).
label	Is the label of the executable statement that receives control if an error occurs.
i-var	Is a scalar integer variable that is defined as a positive integer if an error occurs, and as zero if no error occurs (see I/O Status Specifier).

Example

In the following example, the FIND statement positions logical unit 1 at the first record in the file. The file's associated variable is set to one:

```
FIND(1, REC=1)
```

In the following example, the FIND statement positions the file at the record identified by the content of INDX. The file's associated variable is set to the value of INDX:

```
FIND(4, REC=INDX)
```

See Also

- E to F
- Forms for Direct-Access READ Statements
- I/O Control List

FINDFILEQQ

Portability Function: *Searches for a specified file in the directories listed in the path contained in the environment variable.*

Module

USE IFPORT

Syntax

`result = FINDFILEQQ (filename,varname,pathbuf)`

filename	(Input) Character*(*). Name of the file to be found.
varname	(Input) Character*(*). Name of an environment variable containing the path to be searched.
pathbuf	(Output) Character*(*). Buffer to receive the full path of the file found.

Results

The result type is INTEGER(4). The result is the length of the string containing the full path of the found file returned in *pathbuf*, or 0 if no file is found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

CHARACTER(256) pathname

INTEGER(4) pathlen

pathlen = FINDFILEQQ("libfmt.lib", "LIB", pathname)

WRITE (*,*) pathname

END
```

See Also

- E to F
- FULLPATHQQ

- GETFILEINFOQQ
- SPLITPATHQQ

FIRSTPRIVATE

Parallel Directive Clause: *Provides a superset of the functionality provided by the PRIVATE clause; objects are declared PRIVATE and they are initialized with certain values.*

Syntax

```
FIRSTPRIVATE (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /). Variables that appear in a FIRSTPRIVATE list are subject to PRIVATE clause semantics. In addition, private (local) copies of each variable in the different threads are initialized to the value the variable had before the parallel region started.

See Also

- E to F
- PRIVATE clause

Building Applications: Debugging Shared Variables

Optimizing Applications: PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses

Optimizing Applications: Worksharing Construct Directives

FIXEDFORMLINESIZE

General Compiler Directive: *Sets the line length for fixed-form Fortran source code.*

Syntax

```
cDEC$ FIXEDFORMLINESIZE:{72 | 80 | 132}
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

You can set FIXEDFORMLINESIZE to 72 (the default), 80, or 132 characters. The FIXEDFORMLINESIZE setting remains in effect until the end of the file, or until it is reset.

The FIXEDFORMLINESIZE directive sets the source-code line length in include files, but not in USE modules, which are compiled separately. If an include file resets the line length, the change does not affect the host file.

This directive has no effect on free-form source code.

Example

```
cDEC$ NOFREEFORM
cDEC$ FIXEDFORMLINESIZE:132
      WRITE (*,*) 'Sentence that goes beyond the 72nd column without continuation.'
```

See Also

- E to F
- FREEFORM and NOFREEFORM
- Source Forms
- General Compiler Directives

Building Applications: /fixed

Building Applications: Compiler Directives Related to Options

FLOAT

Elemental Intrinsic Function (Generic):
Converts an integer to REAL(4).

See Also

- E to F
- REAL

FLOODFILL, FLOODFILL_W (W*32, W*64)

Graphics Functions: *Fill an area using the current color index and fill mask.*

Module

USE IFQWIN

Syntax

```
result = FLOODFILL (x,y,bcolor)
result = FLOODFILL_W (wx,wy,bcolor)
```

x, y	(Input) INTEGER(2). Viewport coordinates for fill starting point.
bcolor	(Input) INTEGER(2). Color index of the boundary color.
wx, wy	(Input) REAL(8). Window coordinates for fill starting point.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *bcolor*, or if the starting point lies outside the clipping region).

FLOODFILL begins filling at the viewport-coordinate point (*x*, *y*). FLOODFILL_W begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by FLOODFILL and FLOODFILL_W is set by SETCOLOR. You can obtain the current fill color index by calling GETCOLOR. These functions allow access only to the colors in the palette (256 or less). To access all available colors on a VGA (262,144 colors) or a true color system, use the RGB functions FLOODFILLRGB and FLOODFILLRGB_W.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current graphics color index set by SETCOLOR. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *bcolor*.

NOTE. The FLOODFILL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the FloodFill routine by including the IFWIN module, you need to specify the routine name as MSFWIN$FloodFill. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) status, bcolor, red, blue
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
x2 = 240; y2 = 150
red = 4
blue = 1
status = SETCOLOR(red)
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
bcolor = GETCOLOR()
status = SETCOLOR (blue)
xinterior = 160; yinterior = 100
status = FLOODFILL (xinterior, yinterior, bcolor)
END
```

See Also

- E to F
- FLOODFILLRGB, FLOODFILLRGB_W
- ELLIPSE
- GETCOLOR
- GETFILLMASK
- GRSTATUS
- PIE
- SETCLIPRGN
- SETCOLOR
- SETFILLMASK

Building Applications: Setting Figure Properties

FLOODFILLRGB, FLOODFILLRGB_W (W*32, W*64)

Graphics Functions: *Fill an area using the current Red-Green-Blue (RGB) color and fill mask.*

Module

USE IFQWIN

Syntax

```
result = FLOODFILLRGB (x,y,color)
result = FLOODFILLRGB_W (wx,wy,color)
```

x, y	(Input) INTEGER(2). Viewport coordinates for fill starting point.
color	(Input) INTEGER(4). RGB value of the boundary color.
wx, wy	(Input) REAL(8). Window coordinates for fill starting point.

Results

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, 0 (occurs if the fill could not be completed, or if the starting point lies on a pixel with the boundary color *color*, or if the starting point lies outside the clipping region).

FLOODFILLRGB begins filling at the viewport-coordinate point (*x*, *y*). FLOODFILLRGB_W begins filling at the window-coordinate point (*wx*, *wy*). The fill color used by FLOODFILLRGB and FLOODFILLRGB_W is set by SETCOLORRGB. You can obtain the current fill color by calling GETCOLORRGB.

If the starting point lies inside a figure, the interior is filled; if it lies outside a figure, the background is filled. In both cases, the fill color is the current color set by SETCOLORRGB. The starting point must be inside or outside the figure, not on the figure boundary itself. Filling occurs in all directions, stopping at pixels of the boundary color *color*.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) status
INTEGER(4) result, bcolor
INTEGER(2) x1, y1, x2, y2, xinterior, yinterior
x1 = 80; y1 = 50
x2 = 240; y2 = 150
result = SETCOLORRGB(Z'008080') ! red
status = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
bcolor = GETCOLORRGB( )
result = SETCOLORRGB (Z'FF0000') ! blue
xinterior = 160; yinterior = 100
result = FLOODFILLRGB (xinterior, yinterior, bcolor)
END
```

See Also

- E to F
- ELLIPSE
- FLOODFILL
- GETCOLORRGB
- GETFILLMASK
- GRSTATUS
- PIE
- SETCLIPRGN
- SETCOLORRGB
- SETFILLMASK

Building Applications: Setting Figure Properties

FLOOR

Elemental Intrinsic Function (Generic):
Returns the greatest integer less than or equal to its argument.

Syntax

```
result = FLOOR (a[,kind])
```

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer initialization expression. This argument is a Fortran 95 feature.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is equal to the greatest integer less than or equal to *a*.

The setting of compiler options specifying integer size can affect this function.

Example

FLOOR (4.8) has the value 4.

FLOOR (-5.6) has the value -6.

The following shows another example:

```
I = FLOOR(3.1)  ! returns 3

I = FLOOR(-3.1) ! returns -4
```

See Also

- E to F
- CEILING

FLUSH Directive

OpenMP* Fortran Compiler Directive: *Identifies synchronization points at which the implementation must provide a consistent view of memory.*

Syntax

```
c$OMP FLUSH [(list)]
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
list	Is the name of one or more variables to be flushed. Names must be separated by commas.

The FLUSH directive must appear at the precise point in the code at which the synchronization is required. To avoid flushing all variables, specify a *list*.

Thread-visible variables are written back to memory at the point at which this directive appears. Modifications to thread-visible variables are visible to all threads after this point. Subsequent reads of thread-visible variables fetch the latest copy of the data.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

The FLUSH directive is implied for the following directives (unless the NOWAIT keyword is used):

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE

- ORDERED and END ORDERED
- PARALLEL and END PARALLEL
- PARALLEL DO and END PARALLEL DO
- PARALLEL SECTIONS and END PARALLEL SECTIONS

Example

The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM( )
      ISYNC(IAM) = 0
c$OMP BARRIER
      CALL WORK( )
C I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
      ISYNC(IAM) = 1
c$OMP FLUSH(ISYNC)
C WAIT TILL NEIGHBOR IS DONE
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
c$OMP FLUSH(ISYNC)
      END DO
c$OMP END PARALLEL
```

See Also

- E to F
- OpenMP Fortran Compiler Directives

FLUSH Statement

Statement: *Causes data written to a file to become available to other processes or causes data written to a file outside of Fortran to be accessible to a READ statement. It takes one of the following forms:*

Syntax

`FLUSH([UNIT=]`*`io-unit`* `[,ERR=`*`label`*`] [IOSTAT=`*`i-var`*`])`

`FLUSH` *`io-unit`*

`io-unit`	(Input) Is an external unit specifier.
`label`	(Input) Is the label of the branch target statement that receives control if an error occurs.
`i-var`	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

The FLUSH statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

This statement has no effect on file position.

FLUSH Subroutine

Portability Subroutine: *Flushes the contents of an external unit buffer into its associated file.*

Module

USE IFPORT

Syntax

`CALL FLUSH (`*`lunit`*`)`

`lunit`	(Input) INTEGER(4). Number of the external unit to be flushed. Must be currently connected to a file when the subroutine is called. This routine is thread-safe, and locks the associated stream before I/O is performed.

NOTE. The flush is performed in a non-blocking mode. In this mode, the command may return before the physical write is completed. If you want to use a blocking mode of FLUSH use COMMITQQ.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- E to F
- COMMITQQ

FOCUSQQ (W*32, W*64)

QuickWin Function: *Sets focus to the window with the specified unit number.*

Module

USE IFQWIN

Syntax

```
result = FOCUSQQ (iunit)
```

iunit	(Input) INTEGER(4). Unit number of the window to which the focus is set. Unit numbers 0, 5, and 6 refer to the default startup window.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

Units 0, 5, and 6 refer to the default window only if the program does not specifically open them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

Unlike SETACTIVEQQ, FOCUSQQ brings the specified unit to the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when on I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- E to F
- SETACTIVEQQ
- INQFOCUSQQ

Building Applications: Using QuickWin Overview

Building Applications: Giving a Window Focus and Setting the Active Window

FOR_DESCRIPTOR_ASSIGN (W*32, W*64)

Run-Time Subroutine: *Creates an array descriptor in memory.*

Module

USE IFCORE

Syntax

```
CALL FOR_DESCRIPTOR_ASSIGN (dp,base,size,reserved,rank,dims_info)
```

dp	(Input) A Fortran 95/90 pointer to an array; the array can be of any data type.
base	(Input) INTEGER(4) or INTEGER(8). The base address of the data being described by *dp*. Note that a Fortran 95/90 pointer describes both the location and type of the data item.

size (Input) INTEGER(4). The size of the data type; for example, 4 for INTEGER(4).

reserved (Input) INTEGER(4). A logical bitwise OR combination of the following constants, which are defined in `IFCORE.F90`:

- FOR_DESCRIPTOR_ARRAY_DEFINED - Specifies whether the array pointed to has been allocated or associated. If the bit is set, the array has been allocated or associated.
- FOR_DESCRIPTOR_ARRAY_NODEALLOC - Specifies whether the array points to something that can be deallocated by a call to DEALLOCATE, or whether it points to something that cannot be deallocated. For example:

```
integer, pointer :: p(:)
integer, target :: t
p => t   ! t cannot be deallocated
allocate(p(10)) ! t can be deallocated
```

  If the bit is set, the array cannot be deallocated.
- FOR_DESCRIPTOR_ARRAY_CONTIGUOUS - Specifies whether the array pointed to is completely contiguous in memory or whether it is a slice that is not contiguous. If the bit is set, the array is contiguous.

rank (Input) INTEGER(4). The rank of the array pointed to.

dims_info (Input) An array of derived type `FOR_DIMS_INFO`; you must specify a rank for this array. The derived type `FOR_DIMS_INFO` is defined in IFCORE.F90 as follows:

```
TYPE FOR_DIMS_INFO
  INTEGER(4) LOWERBOUND  !Lower bound for the dimension

  INTEGER(4) UPPERBOUND  !Upper bound for the dimension
  INTEGER(4) STRIDE      !Stride for the dimension

END TYPE FOR_DIMS_INFO
```

The FOR_DESCRIPTOR_ASSIGN routine is similar to a Fortran 95/90 pointer assignment, but gives you more control over the assignment, allowing, for example, assignment to any location in memory.

You can also use this routine to create an array that can be used from both Fortran or C.

Example

```
use IFCORE
common/c_array/ array
real(8) array(5,5)
external  init_array
external  c_print_array
real(8),pointer :: p_array(:,:)
type(FOR_DIMS_INFO) dims_info(2)
 call init_array()

 do i=1,5
   do j=1,5
      print *,i,j, array(i,j)
   end do
 end do

 dims_info(1)%LOWERBOUND = 11

 dims_info(1)%UPPERBOUND = 15
 dims_info(1)%STRIDE = 1

 dims_info(2)%LOWERBOUND = -5
 dims_info(2)%UPPERBOUND = -1
 dims_info(2)%STRIDE = 1
```

```
 call FOR_DESCRIPTOR_ASSIGN(p_array, &
   LOC(array), &

   SIZEOF(array(1,1)), &
   FOR_DESCRIPTOR_ARRAY_DEFINED .or. &
   FOR_DESCRIPTOR_ARRAY_NODEALLOC .or. &
   FOR_DESCRIPTOR_ARRAY_CONTIGUOUS,  &
   2, &
   dims_info )

p_array = p_array + 1
call c_print_array()
end
```

The following shows the C program containing init_array and c_print_array:

```
#include <stdio.h>
#if !defined(_WIN32) && !defined(_WIN64)
#define C_ARRAY c_array_
#define INIT_ARRAY init_array_
#define C_PRINT_ARRAY c_print_array_
#endif
double C_ARRAY[5][5];
void INIT_ARRAY(void);
void C_PRINT_ARRAY(void);
void INIT_ARRAY(void)
{
  int i,j;
  for(i=0;i<5;i++)
```

```
    for(j=0;j<5;j++)
       C_ARRAY[i][j] = j + 10*i;

}
void C_PRINT_ARRAY(void)
{
   int i,j;

   for(i=0;i<5;i++){
     for(j=0;j<5;j++)
         printf("%f ", C_ARRAY[i][j]);
     printf("\n");
   }
}
```

See Also

- E to F
- POINTER - Fortran 95/90

FOR_GET_FPE

Run-Time Function: *Returns the current settings of floating-point exception flags. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

```
result = FOR_GET_FPE( )
```

Results

The result type is INTEGER(4). The return value represents the settings of the current floating-point exception flags. The meanings of the bits are defined in the IFPORT module file.

To set floating-point exception flags after program initialization, use FOR_SET_FPE.

Example

```
USE IFCORE

INTEGER*4 FPE_FLAGS

FPE_FLAGS = FOR_GET_FPE ( )
```

See Also

- E to F
- FOR_SET_FPE option for example

for_rtl_finish_

Run-Time Function: *Cleans up the Fortran run-time environment; for example, flushing buffers and closing files. It also issues messages about floating-point exceptions, if any occur.*

Syntax

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

```
result = for_rtl_finish_ ( )
```

Results

The result is an I/O status value. For information on these status values, see *Building Applications*: Using the IOSTAT Value and Fortran Exit Codes.

To initialize the Fortran run-time environment, use function for_rtl_init_ .

Example

Consider the following C code:

```
int io_status;

int for_rtl_finish_ ( );

io_status = for_rtl_finish_ ( );
```

See Also

- E to F
- for_rtl_init_

for_rtl_init_

Run-Time Subroutine: *Initializes the Fortran run-time environment. It establishes handlers and floating-point exception handling, so Fortran subroutinesprocedures behave the same as when called from a Fortran main program.*

Syntax

This routine should be called from a C main program; it is invoked by default from a Fortran main program.

```
CALL for_rtl_init_ (argcount,actarg)
```

argcount	Is a command-line parameter describing the argument count.
actarg	Is a command-line parameter describing the actual arguments.

To clean up the Fortran run-time environment, use function for_rtl_finish_.

Example

Consider the following C code:

```
int argc;

char **argv;

void for_rtl_init_ (int *, char **);

for_rtl_init_ (&argc, argv);
```

See Also

- E to F
- for_rtl_finish_

FOR_SET_FPE

Run-Time Function: *Sets the floating-point exception flags. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

```
result = FOR_SET_FPE (a)
```

a Must be of type INTEGER(4). It contains bit flags controlling floating-point exception trapping, reporting, and result handling.

Results

The result type is INTEGER(4). The return value represents the previous settings of the floating-point exception flags. The meanings of the bits are defined in the IFCORE module file.

To get the current settings of the floating-point exception flags, use FOR_GET_FPE.

Example

```
USE IFCORE

INTEGER*4 OLD_FPE_FLAGS, NEW_FPE_FLAGS

OLD_FPE_FLAGS = FOR_SET_FPE (NEW_FPE_FLAGS)
```

The following example program is compiled without any `fpe` options; however, it uses calls to `for_set_fpe` to enable the same flags as when compiling with the `fpe:0` option. The new flags can be verified by compiling the program with the `-fpe:0` option.

```
program samplefpe
use ifcore
implicit none
INTEGER(4) :: ORIGINAL_FPE_FLAGS, NEW_FPE_FLAGS
INTEGER(4) :: CURRENT_FPE_FLAGS, PREVIOUS_FPE_FLAGS
NEW_FPE_FLAGS = FPE_M_TRAP_UND + FPE_M_TRAP_OVF + FPE_M_TRAP_DIV0  &
+ FPE_M_TRAP_INV + FPE_M_ABRUPT_UND + FPE_M_ABRUPT_DMZ
ORIGINAL_FPE_FLAGS = FOR_SET_FPE (NEW_FPE_FLAGS)
CURRENT_FPE_FLAGS = FOR_GET_FPE ()
print *,"The original FPE FLAGS were:"
CALL PRINT_FPE_FLAGS(ORIGINAL_FPE_FLAGS)
print *," "
print *,"The new FPE FLAGS are:"
CALL PRINT_FPE_FLAGS(CURRENT_FPE_FLAGS)
!! restore the fpe flag to their original values
PREVIOUS_FPE_FLAGS = FOR_SET_FPE (ORIGINAL_FPE_FLAGS)
end
subroutine PRINT_FPE_FLAGS(fpe_flags)
use ifcore
implicit none
integer(4)   :: fpe_flags
character(3) :: toggle
print 10, fpe_flags, fpe_flags
10 format(X,'FPE FLAGS = 0X',Z8.8," B'",B32.32)
if ( IAND(fpe_flags, FPE_M_TRAP_UND) .ne. 0 ) then
 toggle = "ON"
```

```
else
 toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_UND    :", toggle
if ( IAND(fpe_flags, FPE_M_TRAP_OVF) .ne. 0 ) then
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_OVF    :", toggle
if ( IAND(fpe_flags, FPE_M_TRAP_DIV0) .ne. 0 ) then
 toggle = "ON"
else
```

```
 toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_DIV0   :", toggle
if ( IAND(fpe_flags, FPE_M_TRAP_INV) .ne. 0 ) then
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_TRAP_INV    :", toggle
if ( IAND(fpe_flags, FPE_M_ABRUPT_UND) .ne. 0 ) then
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_UND  :", toggle
if ( IAND(fpe_flags, FPE_M_ABRUPT_OVF) .ne. 0 ) then
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_OVF  :", toggle
if ( IAND(fpe_flags, FPE_M_ABRUPT_DMZ) .ne. 0 ) then
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_DIV0 :", toggle
if ( IAND(fpe_flags, FPE_M_ABRUPT_DIV0) .ne. 0 ) then
```

```
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_INV  :", toggle
if ( IAND(fpe_flags, FPE_M_ABRUPT_DMZ) .ne. 0 ) then ! ABRUPT_DMZ
 toggle = "ON"
else
 toggle = "OFF"
endif
write(*,*) "  FPE_ABRUPT_DMZ  :", toggle, "  (ftz related)"
end subroutine PRINT_FPE_FLAGS
```

The output from this program is as follows:

```
>ifort set_fpe_sample01.f90
>set_fpe_sample01.exe
 The original FPE FLAGS were:
 FPE FLAGS = 0X00000000 B'00000000000000000000000000000000
  FPE_TRAP_UND    :OFF
  FPE_TRAP_OVF    :OFF
  FPE_TRAP_DIV0   :OFF
  FPE_TRAP_INV    :OFF
  FPE_ABRUPT_UND  :OFF
  FPE_ABRUPT_OVF  :OFF
  FPE_ABRUPT_DIV0 :OFF
  FPE_ABRUPT_INV  :OFF
  FPE_ABRUPT_DMZ  :OFF (ftz related)

The new FPE FLAGS are:
 FPE FLAGS = 0X0011000F B'00000000000100010000000000001111
  FPE_TRAP_UND    :ON
  FPE_TRAP_OVF    :ON
  FPE_TRAP_DIV0   :ON
  FPE_TRAP_INV    :ON
  FPE_ABRUPT_UND  :ON
  FPE_ABRUPT_OVF  :OFF
  FPE_ABRUPT_DIV0 :ON
  FPE_ABRUPT_INV  :OFF
  FPE_ABRUPT_DMZ  :ON (ftz related)
```

FOR_SET_REENTRANCY

Run-Time Function: *Controls the type of reentrancy protection that the Fortran Run-Time Library (RTL) exhibits. This routine can be called from a C or Fortran program.*

Module

USE IFCORE

Syntax

`result = FOR_SET_REENTRANCY (mode)`

mode Must be of type INTEGER(4) and contain one of the following options:

- FOR_K_REENTRANCY_NONE Tells the Fortran RTL to perform simple locking around critical sections of RTL code. This type of reentrancy should be used when the Fortran RTL will *not* be reentered due to asynchronous system traps (ASTs) or threads within the application.
- FOR_K_REENTRANCY_ASYNCH Tells the Fortran RTL to perform simple locking and disables ASTs around critical sections of RTL code. This type of reentrancy should be used when the application contains AST handlers that call the Fortran RTL.
- FOR_K_REENTRANCY_THREADED Tells the Fortran RTL to perform thread locking. This type of reentrancy should be used in multithreaded applications.
- FOR_K_REENTRANCY_INFO Tells the Fortran RTL to return the current reentrancy mode.

Results

The result type is INTEGER(4). The return value represents the previous setting of the Fortran Run-Time Library reentrancy mode, unless the argument is FOR_K_REENTRANCY_INFO, in which case the return value represents the current setting.

You must be using an RTL that supports the level of reentrancy you desire. For example, FOR_SET_REENTRANCY ignores a request for thread protection (FOR_K_REENTRANCY_THREADED) if you do not build your program with the thread-safe RTL.

Example

```
PROGRAM SETREENT
USE IFCORE
INTEGER*4    MODE
CHARACTER*10 REENT_TXT(3) /'NONE    ','ASYNCH  ','THREADED'/
PRINT*,'Setting Reentrancy mode to ',REENT_TXT(MODE+1)
MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_NONE)
PRINT*,'Previous Reentrancy mode was ',REENT_TXT(MODE+1)
MODE = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_INFO)
PRINT*,'Current Reentrancy mode is ',REENT_TXT(MODE+1)
END
```

FORALL

Statement and Construct: *The FORALL statement and construct is an element-by-element generalization of the Fortran 95/90 masked array assignment (WHERE statement and construct). It allows more general array shapes to be assigned, especially in construct form.*

Syntax

FORALL is a feature of Fortran 95.

Statement:

```
FORALL (triplet-spec[, triplet-spec] ...[, mask-expr]) assign-stmt
```

Construct:

```
[name:] FORALL (triplet-spec[, triplet-spec] ...[, mask-expr])
   forall-body-stmt
   [forall-body-stmt]...
END FORALL [name]
```

triplet-spec — Is a triplet specification with the following form:

subscript-name= *subscript-1*: *subscript-2*[: *stride*]

The *subscript-name* is a scalar of type integer. It is valid only within the scope of the FORALL; its value is undefined on completion of the FORALL.
The *subscript*s and *stride* cannot contain a reference to any *subscript-name* in *triplet-spec*.
The *stride* cannot be zero. If it is omitted, the default value is 1.
Evaluation of an expression in a triplet specification must not affect the result of evaluating any other expression in another triplet specification.

mask-expr — Is a logical array expression (called the mask expression). If it is omitted, the value .TRUE. is assumed. The mask expression can reference the subscript name in *triplet-spec*.

triplet-spec — Is a triplet specification with the following form:

subscript-name= *subscript-1*: *subscript-2*[: *stride*]

The *subscript-name* is a scalar of type integer. It is valid only within the scope of the FORALL; its value is undefined on completion of the FORALL.
The *subscript*s and *stride* cannot contain a reference to any *subscript-name* in *triplet-spec*.
The *stride* cannot be zero. If it is omitted, the default value is 1.
Evaluation of an expression in a triplet specification must not affect the result of evaluating any other expression in another triplet specification.

Description

If a construct name is specified in the FORALL statement, the same name must appear in the corresponding END FORALL statement.

A FORALL statement is executed by first evaluating all bounds and stride expressions in the triplet specifications, giving a set of values for each subscript name. The FORALL assignment statement is executed for all combinations of subscript name values for which the mask expression is true.

The FORALL assignment statement is executed as if all expressions (on both sides of the assignment) are completely evaluated before any part of the left side is changed. Valid values are assigned to corresponding elements of the array being assigned to. No element of an array can be assigned a value more than once.

A FORALL construct is executed as if it were multiple FORALL statements, with the same triplet specifications and mask expressions. Each statement in the FORALL body is executed completely before execution begins on the next FORALL body statement.

Any procedure referenced in the mask expression or FORALL assignment statement must be pure.

Pure functions can be used in the mask expression or called directly in a FORALL statement. Pure subroutines cannot be called directly in a FORALL statement, but can be called from other pure procedures.

Example

The following example, which is not expressible using array syntax, sets diagonal elements of an array to 1:

```
REAL, DIMENSION(N, N) :: A

FORALL (I=1:N)  A(I, I) = 1
```

Consider the following:

```
FORALL(I = 1:N, J = 1:N, A(I, J) .NE. 0.0) B(I, J) = 1.0 / A(I, J)
```

This statement takes the reciprocal of each nonzero element of array A(1:N, 1:N) and assigns it to the corresponding element of array B. Elements of A that are zero do not have their reciprocal taken, and no assignments are made to corresponding elements of B.

Every array assignment statement and WHERE statement can be written as a FORALL statement, but some FORALL statements cannot be written using just array syntax. For example, the preceding FORALL statement is equivalent to the following:

```
WHERE(A /= 0.0) B = 1.0 / A
```

However, the following FORALL example cannot be written using just array syntax:

```
FORALL(I = 1:N, J = 1:N) H(I, J) = 1.0/REAL(I + J - 1)
```

This statement sets array element H(I, J) to the value 1.0/REAL(I + J - 1) for values of I and J between 1 and N.

Consider the following:

```
TYPE MONARCH
  INTEGER, POINTER :: P
END TYPE MONARCH
TYPE(MONARCH), DIMENSION(8) :: PATTERN
INTEGER, DIMENSION(8), TARGET :: OBJECT
FORALL(J=1:8) PATTERN(J)%P => OBJECT(1+IEOR(J-1,2))
```

This FORALL statement causes elements 1 through 8 of array PATTERN to point to elements 3, 4, 1, 2, 7, 8, 5, and 6, respectively, of OBJECT. IEOR can be referenced here because it is pure.

The following example shows a FORALL construct:

```
FORALL(I = 3:N + 1, J = 3:N + 1)
  C(I, J) = C(I, J + 2) + C(I, J - 2) + C(I + 2, J) + C(I - 2, J)
  D(I, J) = C(I, J)
END FORALL
```

The assignment to array D uses the values of C computed in the first statement in the construct, not the values before the construct began execution.

See Also

- E to F
- WHERE

FORMAT

Statement: *Specifies the form of data being transferred and the data conversion (editing) required to achieve that form.*

Syntax

FORMAT (*format-list*)

format-list	Is a list of one or more of the following edit descriptors, separated by commas or slashes (/):
Data edit descriptors:	I, B, O, Z, F, E, EN, ES, D, G, L, and A.

Control edit descriptors:	T, TL, TR, X, S, SP, SS, BN, BZ, P, :, /, $, \, and Q.
String edit descriptors:	H, 'c', and "c", where *c* is a character constant.

A comma can be omitted in the following cases:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash (/) edit descriptor when the optional repeat specification is not present
- After a slash (/) edit descriptor
- Before or after a colon (:) edit descriptor

Edit descriptors can be nested and a *repeat specification* can precede data edit descriptors, the slash edit descriptor, or a parenthesized list of edit descriptors.

Description

A FORMAT statement must be labeled.

Named constants are not permitted in format specifications.

If the associated I/O statement contains an I/O list, the format specification must contain at least one data edit descriptor or the control edit descriptor Q.

Blank characters can precede the initial left parenthesis, and additional blanks can appear anywhere within the format specification. These blanks have no meaning unless they are within a character string edit descriptor.

When a formatted input statement is executed, the setting of the BLANK specifier (for the relevant logical unit) determines the interpretation of blanks within the specification. If the BN or BZ edit descriptors are specified for a formatted input statement, they supersede the default interpretation of blanks. (For more information on BLANK defaults, see the OPEN statement.

For formatted input, use the comma as an external field separator. The comma terminates the input of fields (for noncharacter data types) that are shorter than the number of characters expected. It can also designate null (zero-length) fields.

The first character of a record transmitted to a line printer or terminal is typically used for carriage control; it is not printed. The first character of such a record should be a blank, 0, 1, $,+, or ASCII NUL. Any other character is treated as a blank.

A format specification cannot specify more output characters than the external record can contain. For example, a line printer record cannot contain more than 133 characters, including the carriage control character.

Whenever an edit descriptor requires an integer constant, you can specify an integer expression in a FORMAT statement. The integer expression must be enclosed by angle brackets (< and >). The following examples are valid format specifications:

```
      WRITE(6,20) INT1
   20 FORMAT(I<MAX(20,5)>)
      WRITE(6,FMT=30) INT2, INT3
   30 FORMAT(I<J+K>, I<2*M>)
```

The integer expression can be any valid Fortran expression, including function calls and references to dummy arguments, with the following restrictions:

- Expressions cannot be used with the H edit descriptor.
- Expressions cannot contain graphical relational operators (such as > and <).

The value of the expression is reevaluated each time an input/output item is processed during the execution of the READ, WRITE, or PRINT statement.

The following tables summarize the different kinds of edit descriptors:

Table 797: Data Edit Descriptors

Code	Form [1]	Effect
A	A[w]	Transfers character or Hollerith values.
B	Bw[.m]	Transfers binary values.
D	Dw.d	Transfers real values with D exponents.
E	Ew.d[Ee]	Transfers real values with E exponents.

Code	Form [1]	Effect
EN	ENw.d[Ee]	Transfers real values with engineering notation.
ES	ESw.d[Ee]	Transfers real values with scientific notation.
F	Fw.d	Transfers real values with no exponent.
G	Gw.d[Ee]	Transfers values of all intrinsic types.
I	Iw[.m]	Transfers decimal integer values.
L	Lw	Transfers logical values: on input, transfers characters; on output, transfers T or F.
O	Ow[.m]	Transfers octal values.
Z	Zw[.m]	Transfers hexadecimal values.

[1] *w* is the field width.

m is the minimum number of digits that must be in the field (including zeros).

d is the number of digits to the right of the decimal point.

E is the exponent field.

e is the number of digits in the exponent.

Table 798: Control Edit Descriptors

Code	Form	Effect
BN	BN	Ignores embedded and trailing blanks in a numeric input field.

Code	Form	Effect
BZ	BZ	Treats embedded and trailing blanks in a numeric input field as zeros.
P	kP	Interprets certain real numbers with a specified scale factor.
Q	Q	Returns the number of characters remaining in an input record.
S	S	Reinvokes optional plus sign (+) in numeric output fields; counters the action of SP and SS.
SP	SP	Writes optional plus sign (+) into numeric output fields.
SS	SS	Suppresses optional plus sign (+) in numeric output fields.
T	Tn	Tabs to specified position.
TL	TLn	Tabs left the specified number of positions.
TR	TRn	Tabs right the specified number of positions.
X	nX	Skips the specified number of positions.
$	$	Suppresses trailing carriage return during interactive I/O.

Code	Form	Effect
:	:	Terminates format control if there are no more items in the I/O list.
/	[r]/	Terminates the current record and moves to the next record.
\	\	Continues the same record; same as $.

Table 799: String Edit Descriptors

Code	Form	Effect
H	nHch[ch...]	Transfers characters following the H edit descriptor to an output record.
'c' [2]	'c'	Transfers the character literal constant (between the delimiters) to an output record.

[2] These delimiters can also be quotation marks (").

Example

```
   INTEGER width, value
   width = 2
   read (*,1) width, value
!  if the input is 3123, prints 123, not 12
1  format ( i1, i<width>)
   print *, value
   END
```

See Also

- E to F
- I/O Formatting
- Format Specifications
- Data Edit Descriptors

FP_CLASS

Elemental Intrinsic Function (Generic):
Returns the class of an IEEE real (S_floating, T_floating, or X_floating) argument. This function cannot be passed as an actual argument.*

Syntax

```
result = FP_CLASS (x)
```

x (Input) Must be of type real.

Results

The result type is INTEGER(4). The return value is one of the following:

Class of Argument	Return Value
Signaling NaN	FOR_K_FP_SNAN
Quiet NaN	FOR_K_FP_QNAN
Positive Infinity	FOR_K_FP_POS_INF
Negative Infinity	FOR_K_FP_NEG_INF
Positive Normalized Number	FOR_K_FP_POS_NORM
Negative Normalized Number	FOR_K_FP_NEG_NORM
Positive Denormalized Number	FOR_K_FP_POS_DENORM
Negative Denormalized Number	FOR_K_FP_NEG_DENORM
Positive Zero	FOR_K_FP_POS_ZERO

Class of Argument	Return Value
Negative Zero	FOR_K_FP_NEG_ZERO

The preceding return values are defined in file `for_fpclass.for`.

Example

FP_CLASS (4.0_8) has the value 4 (FOR_K_FP_POS_NORM).

FPUTC

Portability Function: *Writes a character to the file specified by a Fortran external unit, bypassing normal Fortran input/output.*

Module

USE IFPORT

Syntax

`result = FPUTC (lunit,char)`

lunit	(Input) INTEGER(4). Unit number of a file.
char	(Output) Character*(*). Variable whose value is to be written to the file corresponding to *lunit*.

Results

The result type is INTEGER(4). The result is zero if the write was successful; otherwise, an error code, such as:

EINVAL - The specified unit is invalid (either not already open, or an invalid unit number)

If you use WRITE, READ, or any other Fortran I/O statements with *lunit*, be sure to read *Building Applications*: Input and Output With Portability Routines.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT
integer*4 lunit, i4
character*26 string
character*1 char1
lunit = 1
open (lunit,file = 'fputc.dat')
do i = 1,26
  char1 = char(123-i)
  i4 = fputc(1,char1)        !make valid writes
  if (i4.ne.0) iflag = 1
enddo
rewind (1)
read (1,'(a)') string
print *, string
```

See Also

- E to F
- I/O Formatting

Building Applications: Files, Devices, and Input/Output Hardware

FRACTION

Elemental Intrinsic Function (Generic):
Returns the fractional part of the model representation of the argument value.

Syntax

```
result = FRACTION (x)
```

x (Input) Must be of type real.

Results

The result type is the same as *x*. The result has the value *x** b ^{e}. Parameters *b* and *e* are defined in Model for Real Data. If *x* has the value zero, the result has the value zero.

Example

If 3.0 is a REAL(4) value, FRACTION (3.0) has the value 0.75.

The following shows another example:

```
REAL result

result = FRACTION(3.0) ! returns 0.75

result = FRACTION(1024.0) ! returns 0.5
```

See Also

- E to F
- DIGITS
- RADIX
- EXPONENT
- Data Representation Models

FREE

Intrinsic Subroutine (Specific): *Frees a block of memory that is currently allocated. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL FREE (addr)
```

addr	(Input) Must be of type INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. This value is the starting address of the memory block to be freed, previously allocated by MALLOC.

If the freed address was not previously allocated by MALLOC, or if an address is freed more than once, results are unpredictable.

Example

```
INTEGER(4) SIZE
REAL(4) STORAGE(*)
POINTER (ADDR, STORAGE)     ! ADDR will point to STORAGE
SIZE = 1024                 ! Size in bytes
ADDR = MALLOC(SIZE)         ! Allocate the memory
CALL FREE(ADDR)             ! Free it
```

FREEFORM and NOFREEFORM

General Compiler Directives: *FREEFORM specifies that source code is in free-form format. NOFREEFORM specifies that source code is in fixed-form format.*

Syntax

*c*DEC$ FREEFORM

*c*DEC$ NOFREEFORM

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

When the FREEFORM or NOFREEFORM directives are used, they remain in effect for the remainder of the file, or until the opposite directive is used. When in effect, they apply to include files, but do not affect USE modules, which are compiled separately.

See Also

- E to F
- M to N
- Source Forms
- General Compiler Directives
- free compiler option

Building Applications: Compiler Directives Related to Options

FSEEK

Portability Function: *Repositions a file specified by a Fortran external unit.*

Module

USE IFPORT

Syntax

`result = FSEEK (lunit,offset,from)`

lunit (Input) INTEGER(4). External unit number of a file.

offset (Input) INTEGER(4) or INTEGER(8). Offset in bytes, relative to *from*, that is to be the new location of the file marker.

from (Input) INTEGER(4). A position in the file. It must be one of the following:

Value	Variable	Position
0	SEEK_SET	Positions the file relative to the beginning of the file.
1	SEEK_CUR	Positions the file relative to the current position.
2	SEEK_END	Positions the file relative to the end of the file.

Results

The result type is INTEGER(4). The result is zero if the repositioning was successful; otherwise, an error code, such as:

EINVAL: The specified unit is invalid (either not already open, or an invalid unit number), or the *from* parameter is invalid.

The file specified in *lunit* must be open.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
integer(4) istat, offset, ipos
character ichar
OPEN (unit=1,file='datfile.dat')
offset = 5
ipos = 0
istat=fseek(1,offset,ipos)
if (.NOT. stat) then
  istat=fgetc(1,ichar)
  print *, 'data is ',ichar
end if
```

FSTAT

Portability Function: *Returns detailed information about a file specified by a external unit number.*

Module

USE IFPORT

Syntax

`result = FSTAT (`*`lunit`*`,`*`statb`*`)`

lunit	(Input) INTEGER(4). External unit number of the file to examine.
statb	(Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. The elements of *statb* contain the following values:

Element	Description	Values or Notes
statb(1)	Device the file resides on	W*32, W*64: Always 0 L*X: System dependent
statb(2)	File inode number	W*32, W*64: Always 0 L*X: System dependent
statb(3)	Access mode of the file	See the table in Results
statb(4)	Number of hard links to the file	W*32, W*64: Always 1 L*X: System dependent
statb(5)	User ID of owner	W*32, W*64: Always 1 L*X: System dependent
statb(6)	Group ID of owner	W*32, W*64: Always 1 L*X: System dependent
statb(7)	Raw device the file resides on	W*32, W*64: Always 0 L*X: System dependent
statb(8)	Size of the file	

Element	Description	Values or Notes
statb(9)	Time when the file was last accessed[1]	W*32, W*64: Only available on non-FAT file systems; undefined on FAT systems L*X: System dependent
statb(10)	Time when the file was last modified[1]	
statb(11)	Time of last file status change[1]	W*32, W*64: Same as stat(10) L*X: System dependent
statb(12)	Blocksize for file system I/O operations	W*32, W*64: Always 1 L*X: System dependent

[1]Times are in the same format returned by the TIME function (number of seconds since 00:00:00 Greenwich mean time, January 1, 1970).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, returns an error code equal to EINVAL (*lunit* is not a valid unit number, or is not open).

The access mode (the third element of *statb*) is a bitmap consisting of an IOR of the following constants:

Symbolic name	Constant	Description	Notes
S_IFMT	O'0170000'	Type of file	

Symbolic name	Constant	Description	Notes
S_IFDIR	O'0040000'	Directory	
S_IFCHR	O'0020000'	Character special	Never set on Windows* systems
S_IFBLK	O'0060000'	Block special	Never set on Windows systems
S_IFREG	O'0100000'	Regular	
S_IFLNK	O'0120000'	Symbolic link	Never set on Windows systems
S_IFSOCK	O'0140000'	Socket	Never set on Windows systems
S_ISUID	O'0004000'	Set user ID on execution	Never set on Windows systems
S_ISGID	O'0002000'	Set group ID on execution	Never set on Windows systems
S_ISVTX	O'0001000'	Save swapped text	Never set on Windows systems
S_IRWXU	O'0000700'	Owner's file permissions	
S_IRUSR, S_IREAD	O'0000400'	Owner's read permission	Always true on Windows systems
S_IWUSR, S_IWRITE	O'0000200'	Owner's write permission	
S_IXUSR, S_IEXEC	O'0000100'	Owner's execute permission	Based on file extension (.EXE, .COM, .CMD, or .BAT)

Symbolic name	Constant	Description	Notes
S_IRWXG	O'0000070'	Group's file permissions	Same as S_IRWXU on Windows systems
S_IRGRP	O'0000040'	Group's read permission	Same as S_IRUSR on Windows systems
S_IWGRP	O'0000020'	Group's write permission	Same as S_IWUSR on Windows systems
S_IXGRP	O'0000010'	Group's execute permission	Same as S_IXUSR on Windows systems
S_IRWXO	O'0000007'	Other's file permissions	Same as S_IRWXU on Windows systems
S_IROTH	O'0000004'	Other's read permission	Same as S_IRUSR on Windows systems
S_IWOTH	O'0000002'	Other's write permission	Same as S_IWUSR on Windows systems
S_IXOTH	O'0000001'	Other's execute permission	Same as S_IXUSR on Windows systems

STAT returns the same information as FSTAT, but accesses files by name instead of external unit number.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
integer(4) statarray(12), istat
OPEN (unit=1,file='datfile.dat')
ISTAT = FSTAT (1, statarray)
if (.NOT. istat) then
    print *, statarray
end if
```

See Also

- E to F
- INQUIRE
- STAT

FTELL, FTELLI8

Portability Functions: *Return the current position of a file.*

Module

USE IFPORT

Syntax

`result = FTELL (lunit)`

`result = FTELLI8 (lunit)`

lunit (Input) INTEGER(4). External unit number of a file.

Results

The result type is INTEGER(4) for FTELL; INTEGER(8) for FTELLI8. The result is the offset, in bytes, from the beginning of the file. A negative value indicates an error, which is the negation of the IERRNO error code. The following is an example of an error code:

EINVAL: *lunit* is not a valid unit number, or is not open.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

FULLPATHQQ

Portability Function: *Returns the full path for a specified file or directory.*

Module

USE IFPORT

Syntax

`result = FULLPATHQQ (name,pathbuf)`

name	(Input) Character*(*). Item for which you want the full path. Can be the name of a file in the current directory, a relative directory or file name, or a network uniform naming convention (UNC) path.
pathbuf	(Output) Character*(*). Buffer to receive full path of the item specified in *name*.

Results

The result type is INTEGER(4). The result is the length of the full path in bytes, or 0 if the function fails. This function does not verify that the resulting path and file name are valid nor that they exist.

The length of the full path depends upon how deeply the directories are nested on the drive you are using. If the full path is longer than the character buffer provided to return it (*pathbuf*), FULLPATHQQ returns only that portion of the path that fits into the buffer.

Check the length of the path before using the string returned in *pathbuf*. If the longest full path you are likely to encounter does not fit into the buffer you are using, allocate a larger character buffer. You can allocate the largest possible path buffer with the following statements:

```
USE IFPORT
CHARACTER($MAXPATH) pathbuf
```

$MAXPATH is a symbolic constant defined in `IFPORT.F90` as 260.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
      USE IFPORT
      USE IFCORE
      CHARACTER($MAXPATH) buf
      CHARACTER(3)        drive
      CHARACTER(256)      dir
      CHARACTER(256)      name
      CHARACTER(256)      ext
      CHARACTER(256)      file
      INTEGER(4)          len
      DO WHILE (.TRUE.)
        WRITE (*,*)
        WRITE (*,'(A, \)') ' Enter filename (Hit &
                            RETURN to exit): '
        len = GETSTRQQ(file)
        IF (len .EQ. 0) EXIT
        len = FULLPATHQQ(file, buf)
        IF (len .GT. 0) THEN
          WRITE (*,*) buf(:len)
        ELSE
          WRITE (*,*) 'Can''t get full path'
          EXIT
        END IF
!
!       Split path
        WRITE (*,*)
        len = SPLITPATHQQ(buf, drive, dir, name, ext)
        IF (len .NE. 0) THEN
```

```
          WRITE (*, 900) ' Drive: ', drive
          WRITE (*, 900) ' Directory: ', dir(1:len)
          WRITE (*, 900) ' Name: ', name
          WRITE (*, 900) ' Extension: ', ext
        ELSE
          WRITE (*, *) 'Can''t split path'
        END IF
      END DO
900   FORMAT (A, A)
      END
```

See Also

- E to F
- SPLITPATHQQ

FUNCTION

Statement: *The initial statement of a function subprogram. A function subprogram is invoked in an expression and returns a single value (a function result) that is used to evaluate the expression.*

Syntax

```
[prefix [prefix]] FUNCTION name [([d-arg-list])] [suffix]
   [specification-part]
   [execution-part]
[CONTAINS
   internal-subprogram-part]
END [FUNCTION [name]]
```

prefix (Optional) Is any of the following:

- A data type specifier
- RECURSIVE

Permits direct recursion to occur. If a function is directly recursive and array valued, RESULT must also be specified.

- PURE

  Asserts that the procedure has no side effects.

- ELEMENTAL

  Acts on one array element at a time. This is a restricted form of pure procedure.

At most one of each of the above can be specified. You cannot specify ELEMENTAL and RECURSIVE together. You cannot specify ELEMENTAL if *lang-binding* is specified in *suffix*.

name
Is the name of the function. If RESULT is specified, the function name must not appear in any specification statement in the scoping unit of the function subprogram.
The function name can be followed by the length of the data type. The length is specified by an asterisk (*) followed by any unsigned, nonzero integer that is a valid length for the function's type. For example, REAL FUNCTION LGFUNC*8 (Y, Z) specifies the function result as REAL(8) (or REAL*8).
This optional length specification is not permitted if the length has already been specified following the keyword CHARACTER.

d-arg-list
(Optional) Is a list of one or more dummy arguments.
If there are no dummy arguments and no RESULT variable, the parentheses can be omitted. For example, the following is valid:

```
FUNCTION F
```

suffix
(Optional) Takes one of the following forms:
[RESULT (*r-name*)] *lang-binding*
lang-binding [RESULT (*r-name*)]

r-name
(Optional) Is the name of the function result. This name must not be the same as the function name.

lang-binding
Takes the following form:
BIND (C [, NAME=*ext-name*])

ext-name Is a character scalar initialization expression that can be used to construct the external name.

specification-part Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- PUBLIC and PRIVATE (or their equivalent attributes)

An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.

execution-part Is one or more executable constructs or statements, except for ENTRY or RETURN statements.

internal-subprogram-part Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

The type and kind parameters (if any) of the function's result can be defined in the FUNCTION statement or in a type declaration statement within the function subprogram, but not both. If no type is specified, the type is determined by implicit typing rules in effect for the function subprogram.

Execution begins with the first executable construct or statement following the FUNCTION statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

If you specify CHARACTER*(*), the function assumes the length declared for it in the program unit that invokes it. This type of character function can have different lengths when it is invoked by different program units; it is an obsolescent feature in Fortran 95.

If the length is specified as an integer constant, the value must agree with the length of the function specified in the program unit that invokes the function. If no length is specified, a length of 1 is assumed.

If the function is array-valued or a pointer, the declarations within the function must state these attributes for the function result name. The specification of the function result attributes, dummy argument attributes, and the information in the procedure heading collectively define the interface of the function.

The value of the result variable is returned by the function when it completes execution. Certain rules apply depending on whether the result is a pointer, as follows :

- If the result is a pointer, its allocation status must be determined before the function completes execution. The function must associate a target with the pointer, or cause the pointer to be explicitly disassociated from a target.

  The shape of the value returned by the function is determined by the shape of the result variable when the function completes execution.
- If the result is not a pointer, its value must be defined before the function completes execution. If the result is an array, all the elements must be defined. If the result is a derived-type structure, all the components must be defined.

A function subprogram *cannot* contain a BLOCK DATA statement, a PROGRAM statement, or a MODULE statement. A function can contain SUBROUTINE and FUNCTION statements to define internal procedures. ENTRY statements can be included to provide multiple entry points to the subprogram.

Example

The following example uses the Newton-Raphson iteration method `(F(X) = cosh(X) + cos(X) - A = 0)` to get the root of the function:

```
FUNCTION ROOT(A)
  X  = 1.0
  DO
    EX = EXP(X)
    EMINX = 1./EX
    ROOT  = X - ((EX+EMINX)*.5+COS(X)-A)/((EX-EMINX)*.5-SIN(X))
    IF (ABS((X-ROOT)/ROOT) .LT. 1E-6) RETURN
    X  = ROOT
  END DO
END
```

In the preceding example, the following formula is calculated repeatedly until the difference between X_i and X_{i+1} is less than 1.0E-6:

$$X_{i+1} = X_i - \frac{\cos(X_i) + \cos(X_i) - A}{\sinh(X_i) - \sin(X_i)}$$

The following example shows an assumed-length character function:

```
CHARACTER*(*) FUNCTION REDO(CARG)
  CHARACTER*1 CARG
  DO I=1,LEN(REDO)
    REDO(I:I) = CARG
  END DO
  RETURN
END FUNCTION
```

This function returns the value of its argument, repeated to fill the length of the function.

Within any given program unit, all references to an assumed-length character function must have the same length. In the following example, the REDO function has a length of 1000:

```
CHARACTER*1000 REDO, MANYAS, MANYZS
MANYAS = REDO('A')
MANYZS = REDO('Z')
```

Another program unit within the executable program can specify a different length. For example, the following REDO function has a length of 2:

```
CHARACTER HOLD*6, REDO*2
HOLD = REDO('A')//REDO('B')//REDO('C')
```

The following example shows a dynamic array-valued function:

```
FUNCTION SUB (N)
  REAL, DIMENSION(N) :: SUB
  ...
END FUNCTION
```

The following shows another example:

```
      INTEGER Divby2
10   PRINT *, 'Enter a number'
      READ *, i
      Print *, Divby2(i)
      GOTO 10
      END
C
C     This is the function definition
C
      INTEGER FUNCTION Divby2 (num)
      Divby2=num / 2
      END FUNCTION
```

The following example shows an allocatable function with allocatable arguments:

```
MODULE AP
CONTAINS
 FUNCTION ADD_VEC(P1,P2)
  ! Function to add two allocatable arrays of possibly differing lengths.
  ! The arrays may be thought of as polynomials (coefficients)
   REAL, ALLOCATABLE :: ADD_VEC(:), P1(:), P2(:)
  ! This function returns an allocatable array whose length is set to
  ! the length of the larger input array.
  ALLOCATE(ADD_VEC(MAX(SIZE(P1), SIZE(P2))))
  M = MIN(SIZE(P1), SIZE(P2))
  ! Add up to the shorter input array size
  ADD_VEC(:M) = P1(:M) + P2(:M)
  ! Use the larger input array elements afterwards (from P1 or P2)
  IF(SIZE(P1) > M) THEN
    ADD_VEC(M+1:) = P1(M+1:)
  ELSE IF(SIZE(P2) > M) THEN
    ADD_VEC(M+1:) = P2(M+1:)
  ENDIF
 END FUNCTION
END MODULE
PROGRAM TEST
 USE AP
 REAL, ALLOCATABLE :: P(:), Q(:), R(:), S(:)
 ALLOCATE(P(3))
 ALLOCATE(Q(2))
 ALLOCATE(R(3))
 ALLOCATE(S(3))
 ! Notice that P and Q differ in length
```

```
 P = (/4,2,1/)  ! P = X**2 + 2X + 4
 Q = (/-1,1/)   ! Q =         X - 1
 PRINT *,' Result should be:    3.000000       3.000000       1.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(P, Q)  ! X**2 + 3X + 3
 P = (/1,1,1/)  ! P =  X**2 +  X + 1
 R = (/2,2,2/)  ! R = 2X**2 + 2X + 2
 S = (/3,3,3/)  ! S = 3X**2 + 3X + 3
 PRINT *,' Result should be:    6.000000       6.000000       6.000000'
 PRINT *,' Coefficients are: ', ADD_VEC(ADD_VEC(P,R), S)
END
```

See Also

- E to F
- ENTRY
- SUBROUTINE
- PURE
- ELEMENTAL
- RESULT keyword
- Function References
- Program Units and Procedures
- General Rules for Function and Subroutine Subprograms

G

GERROR

Run-Time Subroutine: *Returns a message for the last error detected by a Fortran run-time routine.*

Module

USE IFCORE

Syntax

CALL GERROR (*string*)

string	(Output) Character*(*). Message corresponding to the last detected error.

The last detected error does not necessarily correspond to the most recent function call. The compiler resets *string* only when another error occurs.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFCORE
character*40 errtext
character char1
integer*4 iflag, i4
. . .!Open unit 1 here
i4=fgetc(1,char1)
if (i4) then
  iflag = 1
  Call GERROR (errtext)
  print *, errtext
end if
```

See Also

- G
- PERROR
- IERRNO

GETACTIVEQQ (W*32, W*64)

QuickWin Function: *Returns the unit number of the currently active child window.*

Module

USE IFQWIN

Syntax

```
result = GETACTIVEQQ( )
```

Results

The result type is INTEGER(4). The result is the unit number of the currently active window. If no child window is active, it returns the parameter QWIN$NOACTIVEWINDOW (defined in `IFQWIN.F90`).

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- G
- SETACTIVEQQ
- GETHWNDQQ

Building Applications: Using QuickWin Overview

Building Applications: Giving a Window Focus and Setting the Active Window

GETARCINFO (W*32, W*64)

Graphics Function: *Determines the endpoints (in viewport coordinates) of the most recently drawn arc or pie.*

Module

USE IFQWIN

Syntax

```
result = GETARCINFO (pstart, pend, ppaint)
```

pstart	(Output) Derived type xycoord. Viewport coordinates of the starting point of the arc.
pend	(Output) Derived type xycoord. Viewport coordinates of the end point of the arc.
ppaint	(Output) Derived type xycoord. Viewport coordinates of the point at which the fill begins.

Results

The result type is INTEGER(2). The result is nonzero if successful. The result is zero if neither the ARC nor the PIE function has been successfully called since the last time CLEARSCREEN or SETWINDOWCONFIG was successfully called, or since a new viewport was selected.

GETARCINFO updates the *pstart* and *pend*xycoord derived types to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to the ARC or PIE functions. The xycoord derived type, defined in IFQWIN.F90, is:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

The returned value in *ppaint* specifies a point from which a pie can be filled. You can use this to fill a pie in a color different from the border color. After a call to GETARCINFO, change colors using SETCOLORRGB. Use the new color, along with the coordinates in *ppaint*, as arguments for the FLOODFILLRGB function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) status, x1, y1, x2, y2, x3, y3, x4, y4
TYPE (xycoord) xystart, xyend, xyfillpt
x1 = 80; y1 = 50
x2 = 240; y2 = 150
x3 = 120; y3 = 80
x4 = 90; y4 = 180
status = ARC(x1, y1, x2, y2, x3, y3, x4, y4)
status = GETARCINFO(xystart, xyend, xyfillpt)
END
```

See Also

- G
- ARC
- FLOODFILLRGB
- GETCOLORRGB
- GRSTATUS
- PIE
- SETCOLORRGB

GETARG

Intrinsic Subroutine: *Returns the specified command-line argument (where the command itself is argument number zero). This subroutine cannot be passed as an actual argument.*

Syntax

```
CALL GETARG (n,buffer[,status])
```

n	(Input) Must be a scalar of type integer. This value is the position of the command-line argument to retrieve. The command itself is argument number 0.

buffer	(Output) Must be a scalar of type default character. Its value is the returned command-line argument.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is the returned completion status. If there were no errors, *status* returns the number of characters in the retrieved command-line argument before truncation or blank-padding. (That is, *status* is the original number of characters in the command-line argument.) Errors return a value of -1. Errors include specifying an argument position less than 0 or greater than the value returned by IARGC.

GETARG returns the *n*th command-line argument. If *n* is zero, the name of the executing program file is returned.

GETARG returns command-line arguments as they were entered. There is no case conversion.

If the command-line argument is shorter than *buffer*, GETARG pads *buffer* on the right with blanks. If the argument is longer than *buffer*, GETARG truncates the argument on the right. If there is an error, GETARG fills *buffer* with blanks.

Example

Assume a command-line invocation of `PROG1 -g -c -a`, and that *buffer* is at least five characters long. The following calls to GETARG return the corresponding arguments in *buffer* and *status*:

Statement	String returned in *buffer*	Length returned in *status*
CALL GETARG (0, buffer, status)	PROG1	5
CALL GETARG (1, buffer)	-g	undefined
CALL GETARG (2, buffer, status)	-c	2
CALL GETARG (3, buffer)	-a	undefined
CALL GETARG (4, buffer, status)	all blanks	-1

See Also

- G
- NARGS
- IARGC
- COMMAND_ARGUMENT_COUNT
- GET_COMMAND
- GET_COMMAND_ARGUMENT

GETBKCOLOR (W*32, W*64)

Graphics Function: *Returns the current background color index for both text and graphics output.*

Module

USE IFQWIN

Syntax

```
result = GETBKCOLOR( )
```

Results

The result type is INTEGER(4). The result is the current background color index.

GETBKCOLOR returns the current background color index for both text and graphics, as set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETBKCOLORRGB, SETCOLORRGB, and SETTEXTCOLORRGB.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are GETBKCOLOR and SETBKCOLOR. The default background index is 0, which is associated with black unless the user remaps the palette with REMAPPALETTERGB.

NOTE. The GETBKCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetBkColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetBkColor. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN

INTEGER(4) bcindex

bcindex = GETBKCOLOR()
```

See Also

- G
- GETBKCOLORRGB
- SETBKCOLOR
- GETCOLOR
- GETTEXTCOLOR
- REMAPALLPALETTERGB, REMAPPALETTERGB

Building Applications: Setting Figure Properties

Building Applications: Using Text Colors

GETBKCOLORRGB (W*32, W*64)

Graphics Function: *Returns the current background Red-Green-Blue (RGB) color value for both text and graphics.*

Module

USE IFQWIN

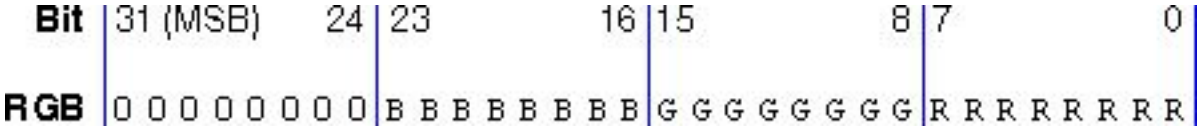
Syntax

```
result = GETBKCOLORRGB( )
```

Results

The result type is INTEGER(4). The result is the RGB value of the current background color for both text and graphics.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with SETBKCOLORRGB. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB and returned with GETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB and returned with GETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETBKCOLOR, SETCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(4) back, fore, oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
oldcolor = SETCOLORRGB(Z'FF') ! red
! reverse the screen
back = GETBKCOLORRGB()
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)
oldcolor = SETCOLORRGB(back)
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

See Also

- G
- GETCOLORRGB
- GETTEXTCOLORRGB
- SETBKCOLORRGB
- GETBKCOLOR

Building Applications: Setting Figure Properties

Building Applications: Using Text Colors

GETC

Portability Function: *Reads the next available character from external unit 5, which is normally connected to the console.*

Module

USE IFPORT

Syntax

`result = GETC (char)`

char	(Output) Character*(*). First character typed at the keyboard after the call to GETC. If unit 5 is connected to a console device, then no characters are returned until the Enter key is pressed.

Results

The result type is INTEGER(4). The result is zero if successful, or -1 if an end-of-file was detected.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAHICS WINDOWS DLL LIB

Example

```
    use IFPORT
    character ans,errtxt*40
    print *, 'Enter a character: '
    ISTAT = GETC (ans)
  if (istat) then
    call gerror(errtxt)
  end if
```

See Also

- G
- GETCHARQQ
- GETSTRQQ

GETCHARQQ

Run-Time Function: *Returns the next keystroke.*

Module

USE IFCORE

Syntax

```
result = GETCHARQQ( )
```

Results

The result type is character with length 1. The result is the character representing the key that was pressed. The value can be any ASCII character.

If the key pressed is represented by a single ASCII character, GETCHARQQ returns the character. If the key pressed is a function or direction key, a hex Z'00' or Z'E0' is returned. If you need to know which function or direction was pressed, call GETCHARQQ a second time to get the extended code for the key.

If there is no keystroke waiting in the keyboard buffer, GETCHARQQ waits until there is one, and then returns it. Compare this to the function PEEKCHARQQ, which returns .TRUE. if there is a character waiting in the keyboard buffer, and .FALSE. if not. You can use PEEKCHARQQ to determine if GETCHARQQ should be called. This can prevent a program from hanging while GETCHARQQ waits for a keystroke that isn't there. Note that PEEKCHARQQ is only supported in console applications.

If your application is a QuickWin or Standard Graphics application, you may want to put a call to PASSDIRKEYSQQ in your program. This will enable the program to get characters that would otherwise be trapped. These extra characters are described in PASSDIRKEYSQQ.

Note that the GETCHARQQ routine used in a console application is a different routine than the one used in a QuickWin or Standard Graphics application. The GETCHARQQ used with a console application does not trap characters that are used in QuickWin for a special purpose, such as scrolling. Console applications do not need, and cannot use PASSDIRKEYSQQ.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
!  Program to demonstrate GETCHARQQ
USE IFCORE
CHARACTER(1) key / 'A' /
PARAMETER (ESC = 27)
PARAMETER (NOREP = 0)
WRITE (*,*) ' Type a key: (or q to quit)'
!  Read keys until ESC or q is pressed
DO WHILE (ICHAR (key) .NE. ESC)
   key = GETCHARQQ()
!  Some extended keys have no ASCII representation
   IF(ICHAR(key) .EQ. NOREP) THEN
     key = GETCHARQQ()
     WRITE (*, 900) 'Not ASCII. Char = NA'
     WRITE (*,*)
!  Otherwise, there is only one key
   ELSE
     WRITE (*,900) 'ASCII. Char = '
     WRITE (*,901) key
   END IF
   IF (key .EQ. 'q' ) THEN
     EXIT
   END IF
   END DO
900   FORMAT (1X, A, \)
901   FORMAT (A)
END
```

See Also

- G
- PEEKCHARQQ
- GETSTRQQ
- INCHARQQ
- MBINCHARQQ
- GETC
- FGETC
- PASSDIRKEYSQQ

GETCOLOR (W*32, W*64)

Graphics Function: *Returns the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = GETCOLOR( )
```

Results

The result type is INTEGER(2). The result is the current color index, if successful; otherwise, -1.

GETCOLOR returns the current color index used for graphics over the background color as set with SETCOLOR. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR and returned with GETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Program to demonstrate GETCOLOR
 PROGRAM COLORS
 USE IFQWIN
 INTEGER(2) loop, loop1, status, color
 LOGICAL(4) winstat
 REAL rnd1, rnd2, xnum, ynum
 type (windowconfig) wc
 status = SETCOLOR(INT2(0))
!  Color random pixels with 15 different colors
 DO loop1 = 1, 15
   color = INT2(MOD(GETCOLOR() +1, 16))
   status = SETCOLOR (color) ! Set to next color
   DO loop = 1, 75
!  Set color of random spot, normalized to be on screen
     CALL RANDOM(rnd1)
     CALL RANDOM(rnd2)
     winstat = GETWINDOWCONFIG(wc)
     xnum = wc%numxpixels
     ynum = wc%numypixels
     status = &
     SETPIXEL(INT2(rnd1*xnum+1),INT2(rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2(rnd2*ynum+1))
     status = &
     SETPIXEL(INT2(rnd1*xnum-1),INT2( rnd2*ynum))
     status = &
     SETPIXEL(INT2(rnd1*xnum),INT2( rnd2*ynum-1))
```

```
  END DO
END DO
END
```

See Also

- G
- GETCOLORRGB
- GETBKCOLOR
- GETTEXTCOLOR
- SETCOLOR

Building Applications: Setting Figure Properties

GETCOLORRGB (W*32, W*64)

Graphics Function: *Returns the current graphics color Red-Green-Blue (RGB) value (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB).*

Module

USE IFQWIN

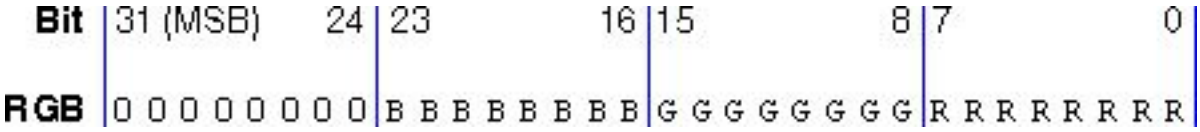
Syntax

```
result = GETCOLORRGB( )
```

Results

The result type is INTEGER(4). The result is the RGB value of the current graphics color.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex FF) the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETCOLORRGB returns the RGB color value of graphics over the background color (used by graphics functions such as ARC, ELLIPSE, and FLOODFILLRGB), set with SETCOLORRGB. GETBKCOLORRGB returns the RGB color value of the current background for both text and graphics, set with SETBKCOLORRGB. GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB.

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) numfonts
INTEGER(4) fore, oldcolor
numfonts = INITIALIZEFONTS ( )
oldcolor = SETCOLORRGB(Z'FF')   ! set graphics
                                !  color to red
fore = GETCOLORRGB()
oldcolor = SETBKCOLORRGB(fore)  ! set background
                                !  to graphics color
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB (Z'FF0000') ! set graphics
                                   !  color to blue
CALL OUTGTEXT("hello, world")
END
```

See Also

- G
- GETBKCOLORRGB
- GETTEXTCOLORRGB
- SETCOLORRGB
- GETCOLOR

Building Applications: Color Mixing

Building Applications: Setting Figure Properties

GET_COMMAND

Intrinsic Subroutine: *Returns the entire command that was used to invoke the program.*

Syntax

```
CALL GET_COMMAND ([command,length,status])
```

command	(Output; optional) Must be a scalar of type default character. If specified, its value is the entire command that was used to invoke the program. If the command cannot be determined, its value is all blanks.
length	(Output; optional) Must be a scalar of type integer. If specified, its value is the significant length of the command that was used to invoke the program. This length includes trailing blanks, but it does not include any truncation or padding used in the command. If the command length cannot be determined, its value is zero.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is -1 if the command argument is present and has a length less than the significant length of the command. If the command cannot be retrieved, its value is positive; otherwise, it is assigned the value zero.

Example

See the example in COMMAND_ARGUMENT_COUNT.

See Also

- G
- GETARG
- NARGS
- IARGC
- COMMAND_ARGUMENT_COUNT
- GET_COMMAND_ARGUMENT

GET_COMMAND_ARGUMENT

Intrinsic Subroutine: *Returns a command line argument of the command that invoked the program. This subroutine cannot be passed as an actual argument.*

Syntax

```
CALL GET_COMMAND_ARGUMENT (number[,value,length,status])
```

number	(Input) Must be a scalar of type integer. It must be non-negative and less than or equal to the value returned by the COMMAND_ARGUMENT_COUNT function. Its value is the position of the command-line argument to retrieve. The command itself is argument number zero.
value	(Output; optional) Must be a scalar of type default character. If specified, its value is the returned command-line argument or all blanks if the value is unknown.
length	(Output; optional) Must be a scalar of type integer. If specified, its value is the length of the returned command-line argument or zero if the length of the argument is unknown. This length includes significant trailing blanks. It does not include any truncation or padding that occurs when the argument is assigned to the *value* argument.
status	(Output; optional) Must be a scalar of type integer. If specified, its value is the returned completion status. It is assigned the value -1 if the value argument is present and has a length less than the significant length of the command argument specified by *number*. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise, it is assigned the value zero.

GET_COMMAND_ARGUMENT returns command-line arguments as they were entered. There is no case conversion.

Example

See the example in COMMAND_ARGUMENT_COUNT.

See Also

- G

- GETARG
- NARGS
- IARGC
- COMMAND_ARGUMENT_COUNT
- GET_COMMAND

GETCONTROLFPQQ

Portability Subroutine: *Returns the floating-point processor control word.*

Module

USE IFPORT

Syntax

CALL GETCONTROLFPQQ (*controlword*)

controlword (Output) INTEGER(2). Floating-point processor control word. The floating-point control word is a bit flag that controls various modes of the floating-point coprocessor.
The control word can be any of the following constants (defined in IFPORT.F90):

Parameter name	Hex value	Description
FPCW$MCW_IC	Z'1000'	**Infinity control mask**
FPCW$AFFINE	Z'1000'	Affine infinity
FPCW$PROJECTIVE	Z'0000'	Projective infinity
FPCW$MCW_PC	Z'0300'	**Precision control mask**
FPCW$64	Z'0300'	64-bit precision
FPCW$53	Z'0200'	53-bit precision
FPCW$24	Z'0000'	24-bit precision

Parameter name	Hex value	Description
FPCW$MCW_RC	Z'0C00'	**Rounding control mask**
FPCW$CHOP	Z'0C00'	Truncate
FPCW$UP	Z'0800'	Round up
FPCW$DOWN	Z'0400'	Round down
FPCW$NEAR	Z'0000'	Round to nearest
FPCW$MCW_EM	Z'003F'	**Exception mask**
FPCW$INVALID	Z'0001'	Allow invalid numbers
FPCW$DENORMAL	Z'0002'	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	Z'0004'	Allow divide by zero
FPCW$OVERFLOW	Z'0008'	Allow overflow
FPCW$UNDERFLOW	Z'0010'	Allow underflow
FPCW$INEXACT	Z'0020'	Allow inexact precision

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0. Exceptions can be disabled by setting the control bits to 1 with SETCONTROLFPQQ.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues.

You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ. If errors on floating-point exceptions are enabled (by clearing the control bits to 0 with SETCONTROLFPQQ), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. Most users do not need to change the default settings. For a full discussion of the floating-point control word, exceptions, and error handling, see *Building Applications*: The Floating-Point Environment Overview.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(2) control

CALL GETCONTROLFPQQ (control)

     !if not rounding down

IF (IAND(control, FPCW$DOWN) .NE. FPCW$DOWN) THEN

  control = IAND(control, NOT(FPCW$MCW_RC)) ! clear all

                                            !  rounding

  control = IOR(control, FPCW$DOWN)         !  set to

                                            !  round down

  CALL SETCONTROLFPQQ(control)

END IF

END
```

See Also

- G
- SETCONTROLFPQQ
- GETSTATUSFPQQ
- SIGNALQQ
- CLEARSTATUSFPQQ

Building Applications: Exception Parameters

Building Applications: Floating-Point Control Word Overview

GETCURRENTPOSITION, GETCURRENTPOSITION_W (W*32, W*64)

Graphics Subroutines: *Return the coordinates of the current graphics position.*

Module

USE IFQWIN

Syntax

```
CALL GETCURRENTPOSITION (t)
CALL GETCURRENTPOSITION_W (wt)
```

t (Output) Derived type `xycoord`. Viewport coordinates of current graphics position. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

wt (Output) Derived type `wxycoord`. Window coordinates of current graphics position. The derived type `wxycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx    ! x-coordinate
  REAL(8) wy    ! y-coordinate
END TYPE wxycoord
```

LINETO, MOVETO, and OUTGTEXT all change the current graphics position. It is in the center of the screen when a window is created.

Graphics output starts at the current graphics position returned by GETCURRENTPOSITION or GETCURRENTPOSITION_W. This position is not related to normal text output (from OUTTEXT or WRITE, for example), which begins at the current text position (see SETTEXTPOSITION). It does, however, affect graphics text output from OUTGTEXT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Program to demonstrate GETCURRENTPOSITION
USE IFQWIN
TYPE (xycoord) position
INTEGER(2)     result
result = LINETO(INT2(300), INT2(200))
CALL GETCURRENTPOSITION( position )
IF (position%xcoord .GT. 50) THEN
  CALL MOVETO(INT2(50), position%ycoord, position)
  WRITE(*,*) "Text unaffected by graphics position"
END IF
result = LINETO(INT2(300), INT2(200))
END
```

See Also

- G
- LINETO
- MOVETO
- OUTGTEXT
- SETTEXTPOSITION
- GETTEXTPOSITION

Building Applications: Setting Graphics Coordinates

GETCWD

Portability Function: *Returns the path of the current working directory.*

Module

USE IFPORT

Syntax

`result = GETCWD (dirname)`

dirname	(Output) Character *(*). Name of the current working directory path, including drive letter.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
    USE IFPORT
    character*30 dirname
!   variable dirname must be long enough to hold entire string
    integer(4) istat
    ISTAT = GETCWD (dirname)
    IF (ISTAT == 0) write *, 'Current directory is ',dirname
```

See Also

- G
- GETDRIVEDIRQQ

GETDAT

Portability Subroutine: *Returns the date.*

Module

USE IFPORT

Syntax

CALL GETDAT (*iyr*, *imon*, *iday*)

iyr	(Output) INTEGER(4) or INTEGER(2). Year (*xxxx*AD).
imon	(Output) INTEGER(4) or INTEGER(2). Month (1-12).
iday	(Output) INTEGER(4) or INTEGER(2). Day of the month (1-31). This subroutine is thread-safe.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
! Program to demonstrate GETDAT and GETTIM
USE IFPORT
INTEGER(4) tmpday, tmpmonth, tmpyear
INTEGER(4) tmphour, tmpminute, tmpsecond, tmphund
CHARACTER(1) mer
CALL GETDAT(tmpyear, tmpmonth, tmpday)
CALL GETTIM(tmphour, tmpminute, tmpsecond, tmphund)
IF (tmphour .GT. 12) THEN
  mer = 'p'
  tmphour = tmphour - 12
ELSE
  mer = 'a'
END IF
WRITE (*, 900) tmpmonth, tmpday, tmpyear
900  FORMAT(I2, '/', I2.2, '/', I4.4)
WRITE (*, 901) tmphour,tmpminute,tmpsecond,tmphund,mer
901  FORMAT(I2, ':', I2.2, ':', I2.2, ':', I2.2, ' ',&
            A, 'm')
END
```

See Also

- G
- GETTIM
- SETDAT
- SETTIM
- DATE portability routine
- FDATE
- IDATE portability routine
- JDATE

GETDRIVEDIRQQ

Portability Function: *Returns the path of the current working directory on a specified drive.*

Module

USE IFPORT

Syntax

`result = GETDRIVEDIRQQ (drivedir)`

drivedir	(Input; output) Character*(*). On input, drive whose current working directory path is to be returned. On output, string containing the current directory on that drive in the form d:\dir.

Results

The result type is INTEGER(4). The result is the length (in bytes) of the full path of the directory on the specified drive. Zero is returned if the path is longer than the size of the character buffer *drivedir*.

You specify the drive from which to return the current working directory by putting the drive letter into *drivedir* before calling GETDRIVEDIRQQ. To make sure you get information about the current drive, put the symbolic constant FILE$CURDRIVE (defined in `IFPORT.F90`) into *drivedir*.

Because drives are identified by a single alphabetic character, GETDRIVEDIRQQ examines only the first letter of *drivedir*. For instance, if *drivedir* contains the path `c:\fps90\bin`, GETDRIVEDIRQQ (*drivedir*) returns the current working directory on drive C and disregards the rest of the path. The drive letter can be uppercase or lowercase.

The length of the path returned depends on how deeply the directories are nested on the drive specified in *drivedir*. If the full path is longer than the length of *drivedir*, GETDRIVEDIRQQ returns only the portion of the path that fits into *drivedir*. If you are likely to encounter a long path, allocate a buffer of size $MAXPATH ($MAXPATH = 260).

On Linux* and Mac OS* X systems, the function gets a path only when symbolic constant FILE$CURDRIVE has been applied to *drivedir*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
!  Program to demonstrate GETDRIVEDIRQQ

USE IFPORT

CHARACTER($MAXPATH) dir

INTEGER(4) length

!  Get current directory

dir = FILE$CURDRIVE

length = GETDRIVEDIRQQ(dir)

IF (length .GT. 0) THEN

  WRITE (*,*) 'Current directory is: '

  WRITE (*,*) dir

ELSE

  WRITE (*,*) 'Failed to get current directory'

END IF

END
```

See Also

- G
- CHANGEDRIVEQQ
- CHANGEDIRQQ
- GETDRIVESIZEQQ
- GETDRIVESQQ
- GETLASTERRORQQ
- SPLITPATHQQ

GETDRIVESIZEQQ

Portability Function: *Returns the total size of the specified drive and space available on it.*

Module

USE IFPORT

Syntax

```
result = GETDRIVESIZEQQ (drive,total,avail)
```

drive	(Input) Character*(*). String containing the letter of the drive to get information about.
total	(Output) INTEGER(4) or INTEGER(4),DIMENSION(2) or INTEGER(8). Total number of bytes on the drive.
avail	(Output) INTEGER(4) or INTEGER(4),DIMENSION(2) or INTEGER(8). Number of bytes of available space on the drive.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The data types and dimension (if any) specified for the *total* and *avail* arguments must be the same. Specifying an array of two INTEGER(4) elements, or an INTEGER(8) argument, allows drive sizes larger than 2147483647 to be returned.

If an array of two INTEGER(4) elements is specified, the least-significant 32 bits are returned in the first element, the most-significant 32 bits in the second element. If an INTEGER(4) scalar is specified, the least-significant 32 bits are returned.

Because drives are identified by a single alphabetic character, GETDRIVESIZEQQ examines only the first letter of *drive*. The drive letter can be uppercase or lowercase. You can use the constant FILE$CURDRIVE (defined in IFPORT.F90) to get the size of the current drive.

If GETDRIVESIZEQQ fails, use GETLASTERRORQQ to determine the reason.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
! Program to demonstrate GETDRIVESQQ and GETDRIVESIZEQQ
USE IFPORT
CHARACTER(26) drives
CHARACTER(1) adrive
LOGICAL(4) status
INTEGER(4) total, avail
INTEGER(2) i
! Get the list of drives
drives = GETDRIVESQQ()
WRITE (*,'(A, A)') ' Drives available: ', drives
!
!Cycle through them for free space and write to console
DO i = 1, 26
  adrive = drives(i:i)
  status = .FALSE.
  WRITE (*,'(A, A, A, \)') ' Drive ', CHAR(i + 64), ':'
  IF (adrive .NE. ' ') THEN
     status = GETDRIVESIZEQQ(adrive, total, avail)
  END IF
  IF (status) THEN
     WRITE (*,*) avail, ' of ', total, ' bytes free.'
  ELSE
     WRITE (*,*) 'Not available'
  END IF
END DO
END
```

See Also

- G
- GETLASTERRORQQ
- GETDRIVESQQ
- GETDRIVEDIRQQ
- CHANGEDRIVEQQ
- CHANGEDIRQQ

GETDRIVESQQ

Portability Function: *Reports which drives are available to the system.*

Module

USE IFPORT

Syntax

```
result = GETDRIVESQQ( )
```

Results

The result type is character with length 26. It is the positional character string containing the letters of the drives available in the system.

The returned string contains letters for drives that are available, and blanks for drives that are not available. For example, on a system with A, C, and D drives, the string 'A CD' is returned.

On Linux* and Mac OS* X systems, the function returns a string filled with spaces.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

See the example for GETDRIVESIZEQQ.

See Also

- G
- GETDRIVEDIRQQ
- GETDRIVESIZEQQ

- CHANGEDRIVEQQ

GETENV

Portability Subroutine: *Returns the value of an environment variable.*

Module

USE IFPORT

Syntax

`CALL GETENV (ename,evalue)`

ename	(Input) Character*(*). Environment variable to search for.
evalue	(Output) Character*(*). Value found for *ename*. Blank if *ename* is not found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT

character*40 libname

CALL GETENV ("LIB",libname)

TYPE *, "The LIB variable points to ",libname
```

See Also

- G
- GETENVQQ

GET_ENVIRONMENT_VARIABLE

Intrinsic Subroutine: *Gets the value of an environment variable.*

Syntax

`CALL GET_ENVIRONMENT_VARIABLE (name [,value,length,status,trim_name])`

name	(Input) Must be a scalar of type default character. It is the name of the environment variable.
value	(Output; optional) Must be a scalar of type default character. If specified, it is assigned the value of the environment variable specified by *name*. If the environment variable does not exist, *value* is assigned all blanks.
length	Must be a scalar of type integer. If specified, its value is the length of the the environment variable, if it exists; otherwise, *length* is set to 0.
status	(Output; optional) Must be a scalar of type integer. If specified, it is assigned a value of 0 if the environment variable exists and either has no value or its value is successfully assigned to value. It is assigned a value of 1 if the environment variable does not exist. For other error conditions, it is assigned a processor-dependent value greater than 2.
trim_name	(Input; optional) Must be a scalar of type logical. If the value is FALSE, then trailing blanks in name are considered significant. Otherwise, they are not considered part of the environment variable's name.

Example

The following program asks for the name of an environment variable. If the environment variable exists in the program's environment, it prints out its value:

```
program print_env_var
character name*20, val*40
integer len, status
write (*,*) 'enter the name of the environment variable'
read (*,*) name
call get_environment_variable (name, val, len, status, .true.)
if (status .ge. 2) then
   write (*,*) 'get_environment_variable failed: status = ', status
   stop
end if
if (status .eq. 1) then
   write (*,*) 'env var does not exist'
   stop
end if
if (status .eq. -1) then
   write (*,*) 'env var length = ', len, ' truncated to 40'
   len = 40
end if
if (len .eq. 0) then
   write (*,*) 'env var exists  but has no value'
   stop
end if
write (*,*) 'env var value = ', val (1:len)
end
```

When the above program is invoked, the following line is displayed:

```
enter the name of the environment variable
```

The following shows an example of what could be displayed if you enter "HOME".

- On a Linux* OS or Mac OS* X system:

```
env var value = /home/our_space/usr4
```

- On a Windows* OS system:

```
env var value = C:/
```

The following shows an example of what could be displayed if you enter "PATH".

- On a Linux OS or Mac OS X system:

```
env var length =          307  truncated to 40
env var value = /site/our_space/usr4/progs/build_area
```

- On a Windows OS system:

```
env var length =          829  truncated to 40
env var value = C:\OUR_SPACE\BUILD_AREA\build_objects\
```

GETENVQQ

Portability Function: *Returns the value of an environment variable.*

Module

USE IFPORT

Syntax

`result = GETENVQQ (varname,value)`

varname	(Input) Character*(*). Name of environment variable.
value	(Output) Character*(*). Value of the specified environment variable, in uppercase.

Results

The result type is INTEGER(4). The result is the length of the string returned in *value*. Zero is returned if the given variable is not defined.

GETENVQQ searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

Note that some environment variables may exist only on a per-process basis and may not be present at the command-line level.

GETENVQQ uses the C runtime routine getenv and SETENVQQ uses the C runtime routine _putenv. From the C documentation:

getenv and _putenv use the copy of the environment pointed to by the global variable _environ to access the environment. getenv operates only on the data structures accessible to the run-time library and not on the environment segment created for the process by the operating system.

In a program that uses the main function, _environ is initialized at program startup to settings taken from the operating system's environment.

Changes made outside the program by the console SET command, for example, SET MY_VAR=ABCDE, will be reflected by GETENVQQ.

GETENVQQ and SETENVQQ will not work properly with the Windows* APIs GetEnvironmentVariable and SetEnvironmentVariable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
!  Program to demonstrate GETENVQQ and SETENVQQ
USE IFPORT
USE IFCORE
INTEGER(4) lenv, lval
CHARACTER(80) env, val, enval
WRITE (*,900) ' Enter environment variable name to create, &
                modify, or delete: '
lenv = GETSTRQQ(env)
IF (lenv .EQ. 0) STOP
WRITE (*,900) ' Value of variable (ENTER to delete): '
lval = GETSTRQQ(val)
IF (lval .EQ. 0) val = ' '
enval = env(1:lenv) // '=' // val(1:lval)
IF (SETENVQQ(enval)) THEN
  lval = GETENVQQ(env(1:lenv), val)
  IF (lval .EQ. 0) THEN
    WRITE (*,*) 'Can''t get environment variable'
  ELSE IF (lval .GT. LEN(val)) THEN
    WRITE (*,*) 'Buffer too small'
  ELSE
    WRITE (*,*) env(:lenv), ': ', val(:lval)
    WRITE (*,*) 'Length: ', lval
  END IF
ELSE
  WRITE (*,*) 'Can''t set environment variable'
END IF
900  FORMAT (A, \)
```

```
END
```

See Also

- G
- SETENVQQ
- GETLASTERRORQQ

GETEXCEPTIONPTRSQQ (i32, i64em; W*32, W*64)

Run-Time Function: *Returns a pointer to C run-time exception information pointers appropriate for use in signal handlers established with SIGNALQQ or direct calls to the C rtl signal( ) routine.*

Module

USE IFCORE

Syntax

```
result = GETEXCEPTIONPTRSQQ( )
```

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture. The return value is the address of a data structure whose members are pointers to exception information captured by the C runtime at the time of an exception. This result value can then be used as the *eptr* argument to routine TRACEBACKQQ to generate a stack trace from a user-defined handler or to inspect the exception context record directly.

Calling GETEXCEPTIONPTRSQQ is only valid within a user-defined handler that was established with SIGNALQQ or a direct call to the C rtl signal( ) function.

For a full description of exceptions and error handling, see *Building Applications*: The Floating-Point Environment Overview.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
PROGRAM SIGTEST
USE IFCORE
...
R3 = 0.0E0
STS = SIGNALQQ(MY_HANDLER)
! Cause a divide by zero exception
R1 = 3.0E0/R3
...
END
INTEGER(4) FUNCTION MY_HANDLER(SIGNUM,EXCNUM)
USE IFCORE
...
EPTRS = GETEXCEPTIONPTRSQQ()
...
CALL TRACEBACKQQ("Application SIGFPE error!",USER_EXIT_CODE=-1,EPTR=EPTRS)
...
MY_HANDLER = 1
END
```

See Also

- G
- TRACEBACKQQ
- GETSTATUSFPQQ
- CLEARSTATUSFPQQ
- SETCONTROLFPQQ
- GETCONTROLFPQQ
- SIGNALQQ

Building Applications: Using SIGNALQQ

GETEXITQQ (W*32, W*64)

QuickWin Function: *Returns the setting for a QuickWin application's exit behavior.*

Module

USE IFQWIN

Syntax

```
result = GETEXITQQ( )
```

Results

The result type is INTEGER(4). The result is exit mode with one of the following constants (defined in `IFQWIN.F90`):

- QWIN$EXITPROMPT - Displays a message box that reads "Program exited with exit status *n*. Exit Window?", where *n* is the exit status from the program.

  If you choose Yes, the application closes the window and terminates. If you choose No, the dialog box disappears and you can manipulate the window as usual. You must then close the window manually.
- QWIN$EXITNOPERSIST - Terminates the application without displaying a message box.
- QWIN$EXITPERSIST - Leaves the application open without displaying a message box.

The default for both QuickWin and Console Graphics applications is QWIN$EXITPROMPT.

Compatibility

STANDARD GRAPHICS QUICKWIN.EXE LIB

Example

```
!  Program to demonstrate GETEXITQQ
     USE IFQWIN
     INTEGER i
     i = GETEXITQQ()
     SELECT CASE (i)
       CASE (QWIN$EXITPROMPT)
         WRITE(*, *) "Prompt on exit."
       CASE (QWIN$EXITNOPERSIST)
         WRITE(*,*) "Exit and close."
       CASE (QWIN$EXITPERSIST)
         WRITE(*,*) "Exit and leave open."
     END SELECT
     END
```

See Also

- G
- SETEXITQQ

Building Applications: Using QuickWin Overview

GETFILEINFOQQ

Portability Function: *Returns information about the specified file. File names can contain wildcards (* and ?).*

Module

USE IFPORT

Syntax

`result = GETFILEINFOQQ (files,buffer,handle)`

files	(Input) Character*(*). Name or pattern of files you are searching for. Can include a full path and wildcards (* and ?).

buffer

(Output) Derived type FILE$INFO or derived type FILE$INFOI8. Information about a file that matches the search criteria in *files*. The derived type FILE$INFO is defined in IFPORT.F90 as follows:

```
TYPE FILE$INFO
  INTEGER(4) CREATION            ! CREATION TIME (-1 on FAT)
  INTEGER(4) LASTWRITE           ! LAST WRITE TO FILE
  INTEGER(4) LASTACCESS          ! LAST ACCESS (-1 on FAT)
  INTEGER(4) LENGTH              ! LENGTH OF FILE
  INTEGER(4) PERMIT              ! FILE ACCESS MODE
  CHARACTER(255) NAME            ! FILE NAME
END TYPE FILE$INFO
```

The derived type FILE$INFOI8 is defined in IFPORT.F90 as follows:

```
TYPE FILE$INFO
  INTEGER(4) CREATION            ! CREATION TIME (-1 on FAT)
  INTEGER(4) LASTWRITE           ! LAST WRITE TO FILE
  INTEGER(4) LASTACCESS          ! LAST ACCESS (-1 on FAT)
  INTEGER(8) LENGTH              ! LENGTH OF FILE
  INTEGER(4) PERMIT              ! FILE ACCESS MODE
  CHARACTER(255) NAME            ! FILE NAME
END TYPE FILE$INFO
```

handle

(Input; output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Control mechanism. One of the following constants, defined in IFPORT.F90:

- FILE$FIRST - First matching file found.
- FILE$LAST - Previous file was the last valid file.
- FILE$ERROR - No matching file found.

Results

The result type is INTEGER(4). The result is the nonblank length of the file name if a match was found, or 0 if no matching files were found.

To get information about one or more files, set the handle to FILE$FIRST and call GETFILEINFOQQ. This will return information about the first file which matches the name and return a handle. If the program wants more files, it should call GETFILEINFOQQ with the handle. GETFILEINFOQQ must be called with the handle until GETFILEINFOQQ sets handle to FILE$LAST, or system resources may be lost.

The derived-type element variables FILE$INFO%CREATION, FILE$INFO%LASTWRITE, and FILE$INFO%LASTACCESS contain packed date and time information that indicates when the file was created, last written to, and last accessed, respectively. To break the time and date into component parts, call UNPACKTIMEQQ. FILE$INFO%LENGTH contains the length of the file in bytes. FILE$INFO%PERMIT contains a set of bit flags describing access information about the file as follows:

Bit flag	Access information for the file
FILE$ARCHIVE	Marked as having been copied to a backup device.
FILE$DIR	A subdirectory of the current directory. Each MS-DOS* directory contains two special files, "." and "..". These are directory aliases created by MS-DOS for use in relative directory notation. The first refers to the current directory, and the second refers to the current directory's parent directory.
FILE$HIDDEN	Hidden. It does not appear in the directory list you request from the command line, the Microsoft* visual development environment browser, or File Manager.
FILE$READONLY	Write-protected. You can read the file, but you cannot make changes to it.
FILE$SYSTEM	Used by the operating system.

Bit flag	Access information for the file
FILE$VOLUME	A logical volume, or partition, on a physical disk drive. This type of file appears only in the root directory of a physical device.

You can use the constant FILE$NORMAL to check that all bit flags are set to 0. If the derived-type element variable FILE$INFO%PERMIT is equal to FILE$NORMAL, the file has no special attributes. The variable FILE$INFO%NAME contains the short name of the file, not the full path of the file.

If an error occurs, call GETLASTERRORQQ to retrieve the error message, such as:

- ERR$NOENT: The file or path specified was not found.
- ERR$NOMEM: Not enough memory is available to execute the command, the available memory has been corrupted, or an invalid block exists, indicating that the process making the call was not allocated properly.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
USE IFPORT
USE IFCORE
  CALL SHOWPERMISSION( )
END
! SUBROUTINE to demonstrate GETFILEINFOQQ
SUBROUTINE SHOWPERMISSION( )
USE IFPORT
  CHARACTER(80) files
  INTEGER(KIND=INT_PTR_KIND( )) handle
  INTEGER(4) length
  CHARACTER(5) permit
  TYPE (FILE$INFO) info
    WRITE (*, 900) ' Enter wildcard of files to view: '
    900 FORMAT (A, \)
    length = GETSTRQQ(files)
    handle = FILE$FIRST
    DO WHILE (.TRUE.)
      length = GETFILEINFOQQ(files, info, handle)
      IF ((handle .EQ. FILE$LAST) .OR. &
          (handle .EQ. FILE$ERROR)) THEN
         SELECT CASE (GETLASTERRORQQ( ))
           CASE (ERR$NOMEM)
             WRITE (*,*) 'Out of memory'
           CASE (ERR$NOENT)
             EXIT
           CASE DEFAULT
             WRITE (*,*) 'Invalid file or path name'
```

```
      END SELECT
    END IF
    permit = ' '
    IF ((info%permit .AND. FILE$HIDDEN) .NE. 0) &
      permit(1:1) = 'H'
    IF ((info%permit .AND. FILE$SYSTEM) .NE. 0) &
      permit(2:2) = 'S'
    IF ((info%permit .AND. FILE$READONLY) .NE. 0) &
      permit(3:3) = 'R'
    IF ((info%permit .AND. FILE$ARCHIVE) .NE. 0) &
      permit(4:4) = 'A'
    IF ((info%permit .AND. FILE$DIR) .NE. 0) &
      permit(5:5) = 'D'
    WRITE (*, 9000) info%name, info%length, permit
    9000 FORMAT (1X, A5, I9, ' ',A6)
  END DO
END SUBROUTINE
```

See Also

- G
- SETFILEACCESSQQ
- SETFILETIMEQQ
- UNPACKTIMEQQ

GETFILLMASK (W*32, W*64)

Graphics Subroutine: *Returns the current pattern used to fill shapes.*

Module

USE IFQWIN

Syntax

```
CALL GETFILLMASK (mask)
```

mask (Output) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element (byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of 0 are unchanged. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(1) style(8). array(8)
INTEGER(2) i
style = 0
style(1) = Z'F'
style(3) = Z'F'
style(5) = Z'F'
style(7) = Z'F'
CALL SETFILLMASK (style)
...
CALL GETFILLMASK (array)
WRITE (*, *) 'Fill mask in bits: '
DO i = 1, 8
  WRITE (*, '(B8)') array(i)
END DO
END
```

See Also

- G
- ELLIPSE
- FLOODFILLRGB
- PIE
- POLYGON
- RECTANGLE
- SETFILLMASK

Building Applications: Setting Figure Properties

GETFONTINFO (W*32, W*64)

Graphics Function: *Returns the current font characteristics.*

Module

USE IFQWIN

Syntax

`result = GETFONTINFO (font)`

font (Output) Derived type FONTINFO. Set of characteristics of the current font. The FONTINFO derived type is defined in IFQWIN.F90 as follows:

```
TYPE FONTINFO
  INTEGER(4) type        ! 1 = truetype, 0 = bit map
  INTEGER(4) ascent      ! Pixel distance from top to
                         !   baseline
  INTEGER(4) pixwidth    ! Character width in pixels,
                         !   0=proportional
  INTEGER(4) pixheight   ! Character height in pixels
  INTEGER(4) avgwidth    ! Average character width in
                         !   pixels
  CHARACTER(81) filename ! File name including path
  CHARACTER(32) facename ! Font name
  LOGICAL(1) italic      ! .TRUE. if current font
                         !   formatted italic
  LOGICAL(1) emphasized  ! .TRUE. if current font
                         !   formatted bold
  LOGICAL(1) underline   ! .TRUE. if current font
                         !   formatted underlined
END TYPE FONTINFO
```

Results

The result type is INTEGER(2). The result is zero if successful; otherwise, -1.

You must initialize fonts with INITIALIZEFONTS before calling any font-related function, including GETFONTINFO.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics

USE IFQWIN

TYPE (FONTINFO) info

INTEGER(2)      numfonts, return, line_spacing

numfonts = INITIALIZEFONTS ( )

return = GETFONTINFO(info)

line_spacing = info%pixheight + 2

END
```

See Also

- G
- GETGTEXTEXTENT
- GETGTEXTROTATION
- GRSTATUS
- OUTGTEXT
- INITIALIZEFONTS
- SETFONT

Building Applications: Using Fonts from the Graphics Library Overview

Building Applications: Setting the Font and Displaying Text

GETGID

Portability Function: *Returns the group ID of the user of a process.*

Module

USE IFPORT

Syntax

```
result = GETGID( )
```

Results

The result type is INTEGER(4). The result corresponds to the primary group of the user under whose identity the program is running. The result is returned as follows:

- On Windows* systems, this function returns the last subauthority of the security identifier for the process. This is unique on a local machine and unique within a domain for domain accounts.

  Note that on Windows systems, domain accounts and local accounts can overlap.
- On Linux* and Mac OS* X systems, this function returns the group identity for the current process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

ISTAT = GETGID( )
```

GETGTEXTEXTENT (W*32, W*64)

Graphics Function: *Returns the width in pixels that would be required to print a given string of text (including any trailing blanks) with OUTGTEXT using the current font.*

Module

USE IFQWIN

Syntax

```
result = GETGTEXTEXTENT (text)
```

text (Input) Character*(*). Text to be analyzed.

Results

The result type is INTEGER(2). The result is the width of *text* in pixels if successful; otherwise, -1 (for example, if fonts have not been initialized with INITIALIZEFONTS).

This function is useful for determining the size of text that uses proportionally spaced fonts. You must initialize fonts with INITIALIZEFONTS before calling any font-related function, including GETGTEXTEXTENT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) status, pwidth
CHARACTER(80) text
status= INITIALIZEFONTS( )
status= SETFONT('t''Arial''h22w10')
pwidth= GETGTEXTEXTENT('How many pixels wide is this?')
WRITE(*,*) pwidth
END
```

See Also

- G
- GETFONTINFO
- OUTGTEXT
- SETFONT
- INITIALIZEFONTS
- GETGTEXTROTATION

Building Applications: Selecting Display Options

GETGTEXTROTATION (W*32, W*64)

Graphics Function: *Returns the current orientation of the font text output by OUTGTEXT.*

Module

USE IFQWIN

Syntax

```
result = GETGTEXTROTATION( )
```

Results

The result type is INTEGER(4). It is the current orientation of the font text output in tenths of degrees. Horizontal is 0 °, and angles increase counterclockwise so that 900 tenths of degrees (90 °) is straight up, 1800 tenths of degrees (180 °) is upside-down and left, 2700 tenths of degrees (270 °) is straight down, and so forth.

The orientation for text output with OUTGTEXT is set with SETGTEXTROTATION.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics

USE IFQWIN

INTEGER ang

REAL rang

ang = GETGTEXTROTATION( )

rang = FLOAT(ang)/10.0

WRITE(*,*) "Text tilt in degrees is: ", rang

END
```

See Also

- G
- OUTGTEXT
- SETFONT

- SETGTEXTROTATION

GETHWNDQQ (W*32, W*64)

QuickWin Function: *Converts a window unit number into a Windows* handle.*

Module

USE IFQWIN

Syntax

`result = GETHWNDQQ (unit)`

unit	(Input) INTEGER(4). The window unit number. If *unit* is set to QWIN$FRAMEWINDOW (defined in `IFQWIN.F90`), the handle of the frame window is returned.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The result is a true Windows handle to the window. If *unit* is not open, it returns -1 .

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- G
- GETACTIVEQQ
- GETUNITQQ
- SETACTIVEQQ

Building Applications: Using QuickWin Overview

Building Applications: Giving a Window Focus and Setting the Active Window

Building Applications: Using Windows API Routines with QuickWin

GETIMAGE, GETIMAGE_W (W*32, W*64)

Graphics Subroutines: *Store the screen image defined by a specified bounding rectangle.*

Module

USE IFQWIN

Syntax

```
CALL GETIMAGE (x1,y1,x2,y2,image)
```

```
CALL GETIMAGE_W (wx1,wy1,wx2,wy2,image)
```

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.
image	(Output) INTEGER(1). Array of single-byte integers. Stored image buffer.

GETIMAGE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). GETIMAGE_W defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

The buffer used to store the image must be large enough to hold it. You can determine the image size by calling IMAGESIZE at run time, or by using the formula described under IMAGESIZE. After you have determined the image size, you can dimension the buffer accordingly.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(1), ALLOCATABLE:: buffer (:)
INTEGER(2) status, x, y, error
INTEGER(4) imsize
x = 50
y = 30
status = ELLIPSE ($GFILLINTERIOR, INT2(x-15), &
                  INT2(y-15), INT2( x+15), INT2(y+15))
imsize = IMAGESIZE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16))
ALLOCATE(buffer (imsize), STAT = error)
IF (error .NE. 0) THEN
  STOP 'ERROR: Insufficient memory'
END IF
CALL GETIMAGE (INT2(x-16), INT2(y-16), &
                  INT2( x+16), INT2(y+16), buffer)
END
```

See Also

- G
- IMAGESIZE
- PUTIMAGE

Building Applications: Transferring Images in Memory

GETLASTERROR

Portability Function: *Returns the last error set.*

Module

USE IFPORT

Syntax

```
result = GETLASTERROR( )
```

Results

The result type is INTEGER(4). The result is the integer corresponding to the last run-time error value that was set.

For example, if you use an ERR= specifier in an I/O statement, your program will not abort if an error occurs. GETLASTERROR provides a way to determine what the error condition was, with a better degree of certainty than just examining `errno`. Your application can then take appropriate action based upon the error number.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

GETLASTERRORQQ

Portability Function: *Returns the last error set by a run-time procedure.*

Module

USE IFPORT

Syntax

```
result = GETLASTERRORQQ( )
```

Results

The result type is INTEGER(4). The result is the most recent error code generated by a run-time procedure.

Library functions that return a logical or integer value sometimes also provide an error code that identifies the cause of errors. GETLASTERRORQQ retrieves the most recent error message. The error constants are defined in `IFPORT.F90`. The following table shows some library routines and the errors each routine produces:

Library routine	Errors produced
BEEPQQ	no error
BSEARCHQQ	ERR$INVAL

Library routine	Errors produced
CHANGEDIRQQ	ERR$NOMEM, ERR$NOENT
CHANGEDRIVEQQ	ERR$INVAL, ERR$NOENT
COMMITQQ	ERR$BADF
DELDIRQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT
DELFILESQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$INVAL
FINDFILEQQ	ERR$NOMEM, ERR$NOENT
FULLPATHQQ	ERR$NOMEM, ERR$INVAL
GETCHARQQ	no error
GETDRIVEDIRQQ	ERR$NOMEM, ERR$RANGE
GETDRIVESIZEQQ	ERR$INVAL, ERR$NOENT
GETDRIVESQQ	no error
GETENVQQ	ERR$NOMEM, ERR$NOENT
GETFILEINFOQQ	ERR$NOMEM, ERR$NOENT, ERR$INVAL
GETLASTERRORQQ	no error
GETSTRQQ	no error
MAKEDIRQQ	ERR$NOMEM, ERR$ACCES, ERR$EXIST, ERR$NOENT
PACKTIMEQQ	no error
PEEKCHARQQ	no error
RENAMEFILEQQ	ERR$NOMEM, ERR$ACCES, ERR$NOENT, ERR$XDEV

Library routine	Errors produced
RUNQQ	ERR$NOMEM, ERR$2BIG, ERR$INVAL, ERR$NOENT, ERR$NOEXEC
SETERRORMODEQQ	no error
SETENVQQ	ERR$NOMEM, ERR$INVAL
SETFILEACCESSQQ	ERR$NOMEM, ERR$INVAL, ERR$ACCES
SETFILETIMEQQ	ERR$NOMEM, ERR$ACCES, ERR$INVAL, ERR$MFILE, ERR$NOENT
SLEEPQQ	no error
SORTQQ	ERR$INVAL
SPLITPATHQQ	ERR$NOMEM, ERR$INVAL
SYSTEMQQ	ERR$NOMEM, ERR$2BIG, ERR$NOENT, ERR$NOEXEC
UNPACKTIMEQQ	no error

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

GETLINESTYLE (W*32, W*64)

Graphics Function: *Returns the current graphics line style.*

Module

USE IFQWIN

Syntax

```
result = GETLINESTYLE( )
```

Results

The result type is INTEGER(2). The result is the current line style.

GETLINESTYLE retrieves the mask (line style) used for line drawing. The mask is a 16-bit number, where each bit represents a pixel in the line being drawn.

If a bit is 1, the corresponding pixel is colored according to the current graphics color and logical write mode; if a bit is 0, the corresponding pixel is left unchanged. The mask is repeated for the entire length of the line. The default mask is Z'FFFF' (a solid line). A dashed line can be represented by Z'FF00' (long dashes) or Z'F0F0' (short dashes).

The line style is set with SETLINESTYLE. The current graphics color is set with SETCOLORRGB or SETCOLOR. SETWRITEMODE affects how the line is displayed.

The line style retrieved by GETLINESTYLE affects the drawing of straight lines as in LINETO, POLYGON and RECTANGLE, but not the drawing of curved lines as in ARC, ELLIPSE or PIE.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as Graphics
     USE IFQWIN
     INTEGER(2) lstyle
     lstyle = GETLINESTYLE()
     WRITE (*, 100) lstyle, lstyle
100  FORMAT (1X, 'Line mask in Hex ', Z4, ' and binary ', B16)
     END
```

See Also

- G
- LINETO
- POLYGON
- RECTANGLE
- SETCOLORRGB
- SETFILLMASK
- SETLINESTYLE

- SETWRITEMODE

Building Applications: Setting Figure Properties

GETLOG

Portability Subroutine: *Returns the user's login name.*

Module

USE IFPORT

Syntax

CALL GETLOG (*name*)

name (Output) Character*(*). User's login name.

The login name must be less than or equal to 64 characters. If the login name is longer than 64 characters, it is truncated. The actual parameter corresponding to *name* should be long enough to hold the login name. If the supplied actual parameter is too short to hold the login name, the login name is truncated.

If the login name is shorter than the actual parameter corresponding to *name*, the login name is padded with blanks at the end, until it reaches the length of the actual parameter.

If the login name cannot be determined, all blanks are returned.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT

character*20 username

CALL GETLOG (username)

print *, "You logged in as ",username
```

GETPHYSCOORD (W*32, W*64)

Graphics Subroutine: *Translates viewport coordinates to physical coordinates.*

Module

USE IFQWIN

Syntax

```
CALL GETPHYSCOORD (x,y,t)
```

x, y	(Input) INTEGER(2). Viewport coordinates to be translated to physical coordinates.
t	(Output) Derived type `xycoord`. Physical coordinates of the input viewport position. The `xycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see *Building Applications*: Understanding Coordinate Systems Overview.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Program to demonstrate GETPHYSCOORD, GETVIEWCOORD,
!  and GETWINDOWCOORD. Build as QuickWin or Standard
!  Graphics
USE IFQWIN
TYPE (xycoord)  viewxy, physxy
TYPE (wxycoord) windxy
CALL SETVIEWPORT(INT2(80), INT2(50), &
                 INT2(240), INT2(150))
!  Get viewport equivalent of point (100, 90)
CALL GETVIEWCOORD (INT2(100), INT2(90), viewxy)
!  Get physical equivalent of viewport coordinates
CALL GETPHYSCOORD (viewxy%xcoord, viewxy%ycoord, &
                   physxy)
!  Get physical equivalent of viewport coordinates
CALL GETWINDOWCOORD (viewxy%xcoord, viewxy%ycoord, &
                     windxy)
!  Write viewport coordinates
WRITE (*,*) viewxy%xcoord, viewxy%ycoord
!  Write physical coordinates
WRITE (*,*) physxy%xcoord, physxy%ycoord
!  Write window coordinates
WRITE (*,*) windxy%wx, windxy%wy
END
```

See Also

- G
- GETVIEWCOORD
- GETWINDOWCOORD

- SETCLIPRGN
- SETVIEWPORT

Building Applications: Setting Graphics Coordinates

GETPID

Portability Function: *Returns the process ID of the current process.*

Module

USE IFPORT

Syntax

```
result = GETPID( )
```

Results

The result type is INTEGER(4). The result is the process ID number of the current process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) istat

istat = GETPID()
```

GETPIXEL, GETPIXEL_W (W*32, W*64)

Graphics Functions: *Return the color index of the pixel at a specified location.*

Module

USE IFQWIN

Syntax

```
result = GETPIXEL (x, y)

result = GETPIXEL_W (wx, wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates for pixel position.
wx, wy	(Input) REAL(8). Window coordinates for pixel position.

Results

The result type is INTEGER(2). The result is the pixel color index if successful; otherwise, -1 (if the pixel lies outside the clipping region, for example).

Color routines without the RGB suffix, such as GETPIXEL, use color indexes, not true color values, and limit you to colors in the palette, at most 256. To access all system colors, use SETPIXELRGB to specify an explicit Red-Green-Blue value and retrieve the value with GETPIXELRGB.

NOTE. The GETPIXEL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetPixel routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetPixel. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- G
- GETPIXELRGB
- GRSTATUS
- REMAPALLPALETTERGB, REMAPPALETTERGB
- SETCOLOR
- GETPIXELS
- SETPIXEL

GETPIXELRGB, GETPIXELRGB_W (W*32, W*64)

Graphics Functions: *Return the Red-Green-Blue (RGB) color value of the pixel at a specified location.*

Module

USE IFQWIN

Syntax

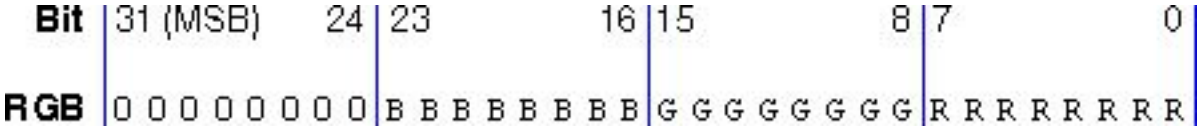
```
result = GETPIXELRGB (x,y)
result = GETPIXELRGB_W (wx,wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates for pixel position.
wx, wy	(Input) REAL(8). Window coordinates for pixel position.

Results

The result type is INTEGER(4). The result is the pixel's current RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETPIXELRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETPIXELRGB returns the true color value of the pixel, set with SETPIXELRGB, SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on the pixel's position and the current configuration of the screen.

SETPIXELRGB (and the other RGB color selection functions SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors to a color value chosen from the entire available range. The non-RGB color functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the

colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) pixcolor, rseed
INTEGER(2) status
REAL rnd1, rnd2
LOGICAL(4) winstat
TYPE (windowconfig) wc
CALL GETTIM (status, status, status, INT2(rseed))
CALL SEED (rseed)
CALL RANDOM (rnd1)
CALL RANDOM (rnd2)
!  Get the color index of a random pixel, normalized to
!    be in the window. Then set current color to that
!    pixel color.
winstat = GETWINDOWCONFIG(wc)
xnum = wc%numxpixels
ynum = wc%numypixels
pixcolor = GETPIXELRGB( INT2( rnd1*xnum ), INT2( rnd2*ynum ))
status = SETCOLORRGB (pixcolor)
END
```

See Also

- G
- SETPIXELRGB
- GETPIXELSRGB
- SETPIXELSRGB
- GETPIXEL, GETPIXEL_W

Building Applications: Color Mixing

GETPIXELS (W*32, W*64)

Graphics Subroutine: *Returns the color indexes of multiple pixels.*

Module

USE IFQWIN

Syntax

`CALL GETPIXELS (n,x,y,color)`

n	(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other arguments.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to get.
color	(Output) INTEGER(2). Array to be filled with the color indexes of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with color indexes of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If the pixel is outside the clipping region, the value placed in the *color* array is undefined. Calls to GETPIXELS with *n* less than 1 are ignored. GETPIXELS is a much faster way to acquire multiple pixel color indexes than individual calls to GETPIXEL.

The range of possible pixel color index values is determined by the current video mode and palette, at most 256 colors. To access all system colors you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function such as SETPIXELSRGB and retrieve the value with GETPIXELSRGB, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- G
- GETPIXELSRGB
- SETPIXELSRGB
- GETPIXEL
- SETPIXELS

GETPIXELSRGB (W*32, W*64)

Graphics Subroutine: *Returns the Red-Green-Blue (RGB) color values of multiple pixels.*

Module

USE IFQWIN

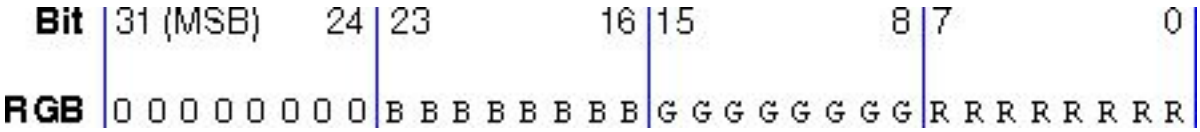
Syntax

```
CALL GETPIXELSRGB (n,x,y,color)
```

n	(Input) INTEGER(4). Number of pixels to get. Sets the number of elements in the other argument arrays.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels.
color	(Output) INTEGER(4). Array to be filled with RGB color values of the pixels at *x* and *y*.

GETPIXELS fills in the array *color* with the RGB color values of the pixels specified by the two input arrays *x* and *y*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the values you retrieve with GETPIXELSRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETPIXELSRGB is a much faster way to acquire multiple pixel RGB colors than individual calls to GETPIXELRGB. GETPIXELSRGB returns an array of true color values of multiple pixels, set with SETPIXELSRGB, SETCOLORRGB, SETBKCOLORRGB, or SETTEXTCOLORRGB, depending on the pixels' positions and the current configuration of the screen.

SETPIXELSRGB (and the other RGB color selection functions SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB) sets colors to a color value chosen from the entire available range. The non-RGB color functions (SETPIXELS, SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) color(50), result
INTEGER(2) x(50), y(50), status
TYPE (xycoord) pos
result = SETCOLORRGB(Z'FF')
CALL MOVETO(INT2(0), INT2(0), pos)
status = LINETO(INT2(100), INT2(200))
!  Get 50 pixels at line 30 in viewport
DO i = 1, 50
  x(i) = i-1
  y(i) = 30
END DO
CALL GETPIXELSRGB(300, x, y, color)
!  Move down 30 pixels and redisplay pixels
DO i = 1, 50
  y(i) = y(i) + 30
END DO
CALL SETPIXELSRGB (50, x, y, color)
END
```

See Also

- G
- SETPIXELSRGB
- GETPIXELRGB, GETPIXELRGB_W
- GETPIXELS
- SETPIXELS

Building Applications: Color Mixing

GETPOS, GETPOSI8

Portability Functions: *Return the current position of a file.*

Module

USE IFPORT

Syntax

`result = GETPOS (lunit)`

`result = GETPOSI8 (lunit)`

lunit	(Input) INTEGER(4). External unit number of a file. The value must be in the range 0 to 100 and the file must be connected.

Results

The result type is INTEGER(4) for GETPOS; INTEGER(8) for GETPOSI8. The result is the offset, in bytes, from the beginning of the file. If an error occurs, the result value is -1 and the following error code is returned in `errno`:

EINVAL: *lunit* is not a valid unit number, or is not open.

These functions are equivalent to FTELL, FTELLI8.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

GETSTATUSFPQQ (W*32, W*64)

Portability Subroutine: *Returns the floating-point processor status word.*

Module

USE IFPORT

Syntax

`CALL GETSTATUSFPQQ (status)`

status	(Output) INTEGER(2). Floating-point processor status word.

The floating-point status word shows whether various floating-point exception conditions have occurred. Intel® Fortran initially clears (sets to 0) all status flags, but after an exception occurs it does not reset the flags before performing additional floating-point operations. A status flag with a value of one thus shows there has been at least one occurrence of the corresponding exception. The following table lists the status flags and their values:

Parameter name	Hex value	Description
FPSW$MSW_EM	Z'003F'	Status Mask (set all flags to 1)
FPSW$INVALID	Z'0001'	An invalid result occurred
FPSW$DENORMAL	Z'0002'	A denormal (very small number) occurred
FPSW$ZERODIVIDE	Z'0004'	A divide by zero occurred
FPSW$OVERFLOW	Z'0008'	An overflow occurred
FPSW$UNDERFLOW	Z'0010'	An underflow occurred
FPSW$INEXACT	Z'0020'	Inexact precision occurred

You can use a logical comparison on the status word returned by GETSTATUSFPQQ to determine which of the six floating-point exceptions listed in the table has occurred.

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0. By default, all exception traps are disabled. Exceptions can be enabled and disabled by clearing and setting the flags with SETCONTROLFPQQ. You can use GETCONTROLFPQQ to determine which exceptions are currently enabled and disabled.

If an exception is disabled, it does not cause an interrupt when it occurs. Instead, floating-point processes generate an appropriate special value (NaN or signed infinity), but the program continues. You can find out which exceptions (if any) occurred by calling GETSTATUSFPQQ.

If errors on floating-point exceptions are enabled (by clearing the flags to 0 with SETCONTROLFPQQ), the operating system generates an interrupt when the exception occurs. By default, these interrupts cause run-time errors, but you can capture the interrupts with SIGNALQQ and branch to your own error-handling routines.

For a full discussion of the floating-point status word, exceptions, and error handling, see *Building Applications*: The Floating-Point Environment Overview.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
!  Program to demonstrate GETSTATUSFPQQ
 USE IFPORT
 INTEGER(2) status
 CALL GETSTATUSFPQQ(status)
!  check for divide by zero
 IF (IAND(status, FPSW$ZERODIVIDE) .NE. 0) THEN
   WRITE (*,*) 'Divide by zero occurred. Look    &
   for NaN or signed infinity in resultant data.'
 END IF
END
```

See Also

- G
- SETCONTROLFPQQ
- GETCONTROLFPQQ
- SIGNALQQ
- CLEARSTATUSFPQQ

GETSTRQQ

Run-Time Function: *Reads a character string from the keyboard using buffered input.*

Module

USE IFCORE

Syntax

`result = GETSTRQQ (buffer)`

buffer	(Output) Character*(*). Character string returned from keyboard, padded on the right with blanks.

Results

The result type is INTEGER(4). The result is the number of characters placed in *buffer*.

The function does not complete until you press Return or Enter.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!  Program to demonstrate GETSTRQQ
 USE IFCORE
 USE IFPORT
 INTEGER(4) length, result
 CHARACTER(80) prog, args
 WRITE (*, '(A, \)') ' Enter program to run: '
 length = GETSTRQQ (prog)
 WRITE (*, '(A, \)') ' Enter arguments: '
 length = GETSTRQQ (args)
 result = RUNQQ (prog, args)
 IF (result .EQ. -1) THEN
   WRITE (*,*) 'Couldn''t run program'
 ELSE
   WRITE (*, '(A, Z4, A)') 'Return code : ', result, 'h'
 END IF
 END
```

See Also

- G
- READ
- GETCHARQQ
- PEEKCHARQQ

GETTEXTCOLOR (W*32, W*64)

Graphics Function: *Returns the current text color index.*

Module

USE IFQWIN

Syntax

```
result = GETTEXTCOLOR( )
```

Results

The result type is INTEGER(2). It is the current text color index.

GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT. The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use SETTEXTCOLORRGB, SETBKCOLORRGB, and SETCOLORRGB.

The default text color index is 15, which is associated with white unless the user remaps the palette.

NOTE. The GETTEXTCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the GetTextColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$GetTextColor. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- G
- OUTTEXT
- REMAPPALETTERGB

- SETCOLOR
- SETTEXTCOLOR

Building Applications: Using Text Colors

GETTEXTCOLORRGB (W*32, W*64)

Graphics Function: *Returns the Red-Green-Blue (RGB) value of the current text color (used with OUTTEXT, WRITE and PRINT).*

Module

USE IFQWIN

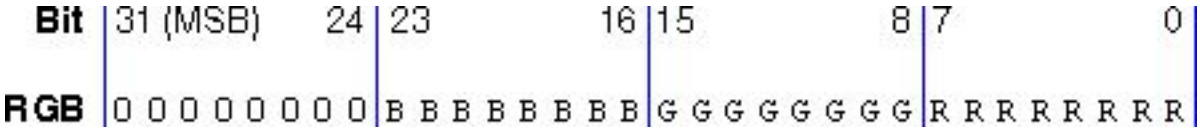
Syntax

```
result = GETTEXTCOLORRGB( )
```

Results

The result type is INTEGER(4). It is the RGB value of the current text color.

In each RGB color calue, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you retrieve with GETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

GETTEXTCOLORRGB returns the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT), set with SETTEXTCOLORRGB. The RGB color value used for graphics is set and returned with SETCOLORRGB and GETCOLORRGB. SETCOLORRGB controls the color used by the graphics function OUTGTEXT, while SETTEXTCOLORRGB controls the color used by all other text output functions. The RGB background color value for both text and graphics is set and returned with SETBKCOLORRGB and GETBKCOLORRGB.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETCOLORRGB) sets the color to a color value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(4) oldtextc, oldbackc, temp
TYPE (rccoord) curpos
!  Save color settings
oldtextc = GETTEXTCOLORRGB()
oldbackc = GETBKCOLORRGB()
CALL CLEARSCREEN( $GCLEARSCREEN )
!  Reset colors
temp = SETTEXTCOLORRGB(Z'00FFFF')  ! full red + full green
                                   !  = full yellow text
temp = SETBKCOLORRGB(Z'FF0000')    ! blue background
CALL SETTEXTPOSITION( INT2(4), INT2(15), curpos)
CALL OUTTEXT( 'Hello, world')
!  Restore colors
temp = SETTEXTCOLORRGB(oldtextc)
temp = SETBKCOLORRGB(oldbackc)
END
```

See Also

- G
- SETTEXTCOLORRGB
- GETBKCOLORRGB
- GETCOLORRGB
- GETTEXTCOLOR

Building Applications: Using Text Colors

GETTEXTPOSITION (W*32, W*64)

Graphics Subroutine: *Returns the current text position.*

Module

USE IFQWIN

Syntax

```
CALL GETTEXTPOSITION (t)
```

t (Output) Derived type `rccord`. Current text position. The derived type `rccoord`is defined in `IFQWIN.F90` as follows:

```
TYPE rccoord
  INTEGER(2) row   ! Row coordinate
  INTEGER(2) col   ! Column coordinate
END TYPE rccoord
```

The text position given by coordinates (1, 1) is defined as the upper-left corner of the text window. Text output from the OUTTEXT function (and WRITE and PRINT statements) begins at the current text position. Font text is not affected by the current text position. Graphics output, including OUTGTEXT output, begins at the current graphics output position, which is a separate position returned by GETCURRENTPOSITION.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics

     USE IFQWIN

     TYPE (rccoord) textpos

     CALL GETTEXTPOSITION (textpos)

     END
```

See Also

- G
- SETTEXTPOSITION
- GETCURRENTPOSITION
- OUTTEXT
- WRITE
- SETTEXTWINDOW

GETTEXTWINDOW (W*32, W*64)

Graphics Subroutine: *Finds the boundaries of the current text window.*

Module

USE IFQWIN

Syntax

`CALL GETTEXTWINDOW (r1,c1,r2,c2)`

r1, c1	(Output) INTEGER(2). Row and column coordinates for upper-left corner of the text window.
r2, c2	(Output) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

Output from OUTTEXT and WRITE is limited to the text window. By default, this is the entire window, unless the text window is redefined by SETTEXTWINDOW.

The window defined by SETTEXTWINDOW has no effect on output from OUTGTEXT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) top, left, bottom, right
DO i = 1, 10
  WRITE(*,*) "Hello, world"
END DO
!  Save text window position
 CALL GETTEXTWINDOW (top, left, bottom, right)
!  Scroll text window down seven lines
 CALL SCROLLTEXTWINDOW (INT2(-7))
!  Restore text window
 CALL SETTEXTWINDOW (top, left, bottom, right)
 WRITE(*,*) "At beginning again"
 END
```

See Also

- G
- GETTEXTPOSITION
- OUTTEXT
- WRITE
- SCROLLTEXTWINDOW
- SETTEXTPOSITION
- SETTEXTWINDOW
- WRAPON

Building Applications: Displaying Character-Based Text

GETTIM

Portability Subroutine: *Returns the time.*

Module

USE IFPORT

Syntax

`CALL GETTIM (ihr, imin, isec, i100th)`

ihr	(Output) INTEGER(4) or INTEGER(2). Hour (0-23).
imin	(Output) INTEGER(4) or INTEGER(2). Minute (0-59).
isec	(Output) INTEGER(4) or INTEGER(2). Second (0-59).
i100th	(Output) INTEGER(4) or INTEGER(2). Hundredths of a second (0-99).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

See the example in GETDAT.

See Also

- G
- GETDAT
- SETDAT
- SETTIM
- CLOCK
- CTIME
- DTIME
- ETIME
- GMTIME
- ITIME
- LTIME
- RTC
- SECNDS portability routine

- TIME portability routine
- TIMEF

GETTIMEOFDAY

Portability Subroutine: *Returns seconds and microseconds since 00:00 Jan 1, 1970.*

Module

USE IFPORT

Syntax

CALL GETTIMEOFDAY (*ret*, *err*)

ret (Output) INTEGER(4). One-dimensional array with 2 elements used to contain numeric time data. The elements of *ret* are returned as follows:

Element	Value
ret(1)	Seconds
ret(2)	Microseconds

err (Output) INTEGER(4).

If an error occurs, *err* contains a value equal to -1 and array *ret* contains zeros.

On Windows* systems, this subroutine has millisecond precision, and the last three digits of the returned value are not significant.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

GETUID

Portability Function: *Returns the user ID of the calling process.*

Module

USE IFPORT

Syntax

```
result = GETUID( )
```

Results

The result type is INTEGER(4). The result corresponds to the user identity under which the program is running. The result is returned as follows:

- On Windows* systems, this function returns the last subauthority of the security identifier for the process. This is unique on a local machine and unique within a domain for domain accounts.

  Note that on Windows systems, domain accounts and local accounts can overlap.
- On Linux* and Mac OS* X systems, this function returns the user identity for the current process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

integer(4) istat

ISTAT = GETUID( )
```

GETUNITQQ (W*32, W*64)

QuickWin Function: *Returns the unit number corresponding to the specified Windows* handle.*

Module

USE IFQWIN

Syntax

```
result = GETUNITQQ (whandle)
```

whandle	(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The Windows handle to the window; this is a unique ID.

Results

The result type is INTEGER(4). The result is the unit number corresponding to the specified Windows handle. If *whandle* does not exist, it returns -1 .

This routine is the inverse of GETHWNDQQ.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- G
- GETHWNDQQ

Building Applications: Using QuickWin Overview

GETVIEWCOORD, GETVIEWCOORD_W (W*32, W*64)

Graphics Subroutines: *Translate physical coordinates or window coordinates to viewport coordinates.*

Module

USE IFQWIN

Syntax

```
CALL GETVIEWCOORD (x, y, t)
```

```
CALL GETVIEWCOORD_W (wx,wy,wt)
```

x, y (Input) INTEGER(2). Physical coordinates to be converted to viewport coordinates.

t (Output) Derived type `xycoord`. Viewport coordinates. The `xycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord   ! x-coordinate
  INTEGER(2) ycoord   ! y-coordinate
END TYPE xycoord
```

wx, wy (Input) REAL(8). Window coordinates to be converted to viewport coordinates.

wt (Output) Derived type `wxycoord`. Window coordinates. The derived type `wxycoord` is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx    ! x-coordinate
  REAL(8) wy    ! y-coordinate
END TYPE wxycoord
```

Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Physical coordinates refer to the whole screen. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see *Building Applications*: Understanding Coordinate Systems Overview.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

See the example program in GETPHYSCOORD.

See Also

- G
- GETPHYSCOORD
- GETWINDOWCOORD

Building Applications: Setting Graphics Coordinates

GETWINDOWCONFIG (W*32, W*64)

QuickWin Function: *Returns the properties of the current window.*

Module

USE IFQWIN

Syntax

```
result = GETWINDOWCONFIG (wc)
```

wc (Output) Derived type `windowconfig`. Contains window properties. The `windowconfig` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE windowconfig

  INTEGER(2) numxpixels            ! Number of pixels on x-axis

  INTEGER(2) numypixels            ! Number of pixels on y-axis

  INTEGER(2) numtextcols            ! Number of text columns
available

  INTEGER(2) numtextrows            ! Number of text rows
available

  INTEGER(2) numcolors              ! Number of color indexes

  INTEGER(4) fontsize           ! Size of default font. Set to

                                !   QWIN$EXTENDFONT when
specifying

                                !   extended attributes, in
which

                                !   case extendfontsize sets
the

                                !   font size

  CHARACTER(80) title               ! The window title

  INTEGER(2) bitsperpixel           ! The number of bits per
pixel

  INTEGER(2) numvideopages          ! Unused

  INTEGER(2) mode                   ! Controls scrolling mode

  INTEGER(2) adapter                ! Unused

  INTEGER(2) monitor                ! Unused

  INTEGER(2) memory                 ! Unused

  INTEGER(2) environment            ! Unused

!

! The next three parameters provide extended font attributes.

!  CHARACTER(32) extendfontname     ! The name of the desired
font
```

```
  INTEGER(4) extendfontsize         ! Takes the same values as
 fontsize,
                                   !   when fontsize is set to
                                   !   QWIN$EXTENDFONT
  INTEGER(4) extendfontattributes   ! Font attributes such as
bold
                                   !   and italic
END TYPE windowconfig
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE. (for example, if there is no active child window).

GETWINDOWCONFIG returns information about the active child window. If you have not set the window properties with SETWINDOWCONFIG, GETWINDOWCONFIG returns default window values.

A typical set of values would be 1024 *x* pixels, 768 *y* pixels, 128 text columns, 48 text rows, and a font size of 8x16 pixels. The resolution of the display and the assumed font size of 8x16 pixels generates the number of text rows and text columns. The resolution (in this case, 1024 *x* pixels by 768 *y* pixels) is the size of the *virtual* window. To get the size of the *physical* window visible on the screen, use GETWSIZEQQ. In this case, GETWSIZEQQ returned the following values: (0,0) for the x and y position of the physical window, 25 for the height or number of rows, and 71 for the width or number of columns.

The number of colors returned depends on the video drive. The window title defaults to "Graphic1" for the default window. All of these values can be changed with SETWINDOWCONFIG.

Note that the bitsperpixel field in the `windowconfig` derived type is an output field only, while the other fields return output values to GETWINDOWCONFIG and accept input values from SETWINDOWCONFIG.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!Build as QuickWin or Standard Graphics App.
USE IFQWIN
LOGICAL(4) status
TYPE (windowconfig) wc
status = GETWINDOWCONFIG(wc)
IF(wc%numtextrows .LT. 10) THEN
  wc%numtextrows = 10
  status = SETWINDOWCONFIG(wc)
  IF(.NOT. status ) THEN  ! if setwindowconfig error
    status = SETWINDOWCONFIG(wc)   ! reset
             ! setwindowconfig with corrected values
    status = GETWINDOWCONFIG(wc)
    IF(wc%numtextrows .NE. 10) THEN
       WRITE(*,*) 'Error: Cannot increase text rows to 10'
    END IF
  END IF
END IF
END
```

See Also

- G
- GETWSIZEQQ
- SETWINDOWCONFIG
- SETACTIVEQQ

Building Applications: Using QuickWin Overview

Building Applications: Accessing Window Properties

Building Applications: Checking the Current Graphics Mode

Building Applications: Setting Graphics Coordinates

GETWINDOWCOORD (W*32, W*64)

Graphics Subroutine: *Converts viewport coordinates to window coordinates.*

Module

USE IFQWIN

Syntax

`CALL GETWINDOWCOORD (x,y,wt)`

x, y	(Input) INTEGER(2). Viewport coordinates to be converted to window coordinates.
wt	(Output) Derived type `wxycoord`. Window coordinates. The `wxycoord` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx   ! x-coordinate
  REAL(8) wy   ! y-coordinate
END TYPE wxycoord
```

Physical coordinates refer to the physical screen. Viewport coordinates refer to an area of the screen defined as the viewport with SETVIEWPORT. Both take integer coordinate values. Window coordinates refer to a window sized with SETWINDOW or SETWSIZEQQ. Window coordinates are floating-point values and allow easy scaling of data to the window area. For a more complete discussion of coordinate systems, see *Building Applications*: Understanding Coordinate Systems Overview.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

See the example program in GETPHYSCOORD.

See Also

- G
- GETCURRENTPOSITION
- GETPHYSCOORD

- GETVIEWCOORD
- MOVETO
- SETVIEWPORT
- SETWINDOW

Building Applications: Setting Graphics Coordinates

GETWRITEMODE (W*32, W*64)

Graphics Function: *Returns the current logical write mode, which is used when drawing lines with the LINETO, POLYGON, and RECTANGLE functions.*

Module

USE IFQWIN

Syntax

```
result = GETWRITEMODE( )
```

Results

The result type is INTEGER(2). The result is the current write mode. Possible return values are:

- $GPSET - Causes lines to be drawn in the current graphics color. (default)
- $GAND - Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- $GOR - Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- $GPRESET - Causes lines to be drawn in the color that is the logical NOT of the current graphics color.
- $GXOR - Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

The default value is $GPSET. These constants are defined in `IFQWIN.F90`.

The write mode is set with SETWRITEMODE.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics App.
   USE IFQWIN
   INTEGER(2) mode
   mode = GETWRITEMODE()
   END
```

See Also

- G
- SETWRITEMODE
- SETLINESTYLE
- LINETO
- POLYGON
- PUTIMAGE
- RECTANGLE
- SETCOLORRGB
- SETFILLMASK
- GRSTATUS

Building Applications: Setting Figure Properties

GETWSIZEQQ (W*32, W*64)

QuickWin Function: *Returns the size and position of a window.*

Module

USE IFQWIN

Syntax

`result = GETWSIZEQQ (unit, ireq, winfo)`

unit	(Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5 and 6 refer to the default startup window only if you have not explicitly opened them with the OPEN statement. To access

information about the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW, defined in IFQWIN.F90.

ireq (Input) INTEGER(4). Specifies what information is obtained. The following symbolic constants, defined in IFQWIN.F90, are available:

- QWIN$SIZEMAX - Gets information about the maximum window size.
- QWIN$SIZECURR - Gets information about the current window size.

winfo (Output) Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type qwinfois defined in IFQWIN.F90 as follows:

```
TYPE QWINFO
  INTEGER(2) TYPE  ! request type (controls
                   !  SETWSIZEQQ)
  INTEGER(2) X     ! x coordinate for upper left
  INTEGER(2) Y     ! y coordinate for upper left
  INTEGER(2) H     ! window height
  INTEGER(2) W     ! window width
END TYPE QWINFO
```

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width returned for a frame window reflects the size in pixels of the client area *excluding* any borders, menus, and status bar at the bottom of the frame window. You should adjust the values used in SETWSIZEQQ to take this into account.

The client area is the area actually available to place child windows.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- G
- GETWINDOWCONFIG
- SETWSIZEQQ

Building Applications: Using QuickWin Overview

Building Applications: Controlling Size and Position of Windows

GMTIME

Portability Subroutine: *Returns the Greenwich mean time in an array of time elements.*

Module

USE IFPORT

Syntax

CALL GMTIME (*stime*,*tarray*)

stime (Input) INTEGER(4). Numeric time data to be formatted. Number of seconds since 00:00:00 Greenwich mean time, January 1, 1970.

tarray (Output) INTEGER(4). One-dimensional array with 9 elements used to contain numeric time data. The elements of *tarray* are returned as follows:

Element	Value
tarray(1)	Seconds (0-61, where 60-61 can be returned for leap seconds)
tarray(2)	Minutes (0-59)
tarray(3)	Hours (0-23)
tarray(4)	Day of month (1-31)

Element	Value
tarray(5)	Month (0-11)
tarray(6)	Number of years since 1900
tarray(7)	Day of week (0-6, where 0 is Sunday)
tarray(8)	Day of year (0-365)
tarray(9)	Daylight saving flag (0 if standard time, 1 if daylight saving time)

CAUTION. This subroutine may cause problems with the year 2000. Use DATE_AND_TIME instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT
integer(4) stime, timearray(9)
! initialize stime to number of seconds since
!   00:00:00 GMT January 1, 1970
stime = time()
CALL GMTIME (stime, timearray)
print *, timearray
end
```

See Also

- G
- DATE_AND_TIME

GOTO - Assigned

Statement: *Transfers control to the statement whose label was most recently assigned to a variable. This feature has been deleted in Fortran 95; it was obsolescent in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.*

Syntax

`GOTO var[[ ,] ( label-list)]`

var	Is a scalar integer variable.
label-list	Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the assigned GO TO statement. The same label can appear more than once in this list.

The variable must have a statement label value assigned to it by an ASSIGN statement (not an arithmetic assignment statement) before the GO TO statement is executed.

If a list of labels appears, the statement label assigned to the variable must be one of the labels in the list.

Both the assigned GO TO statement and its associated ASSIGN statement must be in the same scoping unit.

Example

The following example is equivalent to `GO TO 200`:

```
ASSIGN 200 TO IGO

GO TO IGO
```

The following example is equivalent to `GO TO 450`:

```
ASSIGN 450 TO IBEG

GO TO IBEG, (300,450,1000,25)
```

The following example shows an invalid use of an assigned variable:

```
ASSIGN 10 TO I

J = I

GO TO J
```

In this case, variable J is not the variable assigned to, so it cannot be used in the assigned GO TO statement.

The following shows another example:

```
      ASSIGN 10 TO n
      GOTO n
  10  CONTINUE
```

The following example uses an assigned GOTO statement to check the value of view:

```
C     Show user appropriate view of data depending on
C     security clearance.
      GOTO view (100, 200, 400)
```

See Also

- G
- Obsolescent Features in Fortran 90
- GOTO - Computed GOTO
- GOTO - Unconditional GOTO
- Execution Control

GOTO - Computed

Statement: *Transfers control to one of a set of labeled branch target statements based on the value of an expression. It is an obsolescent feature in Fortran 95.*

Syntax

GOTO (*label-list*) [,] *expr*

label-list	Is a list of labels (separated by commas) of valid branch target statements in the same scoping unit as the computed GO TO statement. (Also called the *transfer list*.) The same label can appear more than once in this list.
expr	Is a scalar numeric expression in the range 1 to n, where *n* is the number of statement labels in *label-list*. If necessary, it is converted to integer data type.

When the computed GO TO statement is executed, the expression is evaluated first. The value of the expression represents the ordinal position of a label in the associated list of labels. Control is transferred to the statement identified by the label. For example, if the list contains (30,20,30,40) and the value of the expression is 2, control is transferred to the statement identified with label 20.

If the value of the expression is less than 1 or greater than the number of labels in the list, control is transferred to the next executable statement or construct following the computed GO TO statement.

Example

The following example shows valid computed GO TO statements:

```
GO TO (12,24,36), INDEX

GO TO (320,330,340,350,360), SITU(J,K) + 1
```

The following shows another example:

```
   next = 1

C

C  The following statement transfers control to statement 10:

C

   GOTO (10, 20) next

   ...

10 CONTINUE

   ...

20 CONTINUE
```

See Also

- G
- obsolescent feature
- GOTO - Unconditional GOTO
- Execution Control

GOTO - Unconditional

Statement: *Transfers control to the same branch target statement every time it executes.*

Syntax

```
GO TO label
```

label	Is the label of a valid branch target statement in the same scoping unit as the GO TO statement.

The unconditional GO TO statement transfers control to the branch target statement identified by the specified label.

Example

The following are examples of GO TO statements:

```
GO TO 7734
GO TO 99999
```

The following shows another example:

```
    integer(2) in
10  print *, 'enter a number from one to ten: '
    read *, in
    select case (in)
    case (1:10)
      exit
    case default
      print *, 'wrong entry, try again'
      goto 10
    end select
```

See Also

- G
- GOTO - Computed GOTO
- Execution Control

GRSTATUS (W*32, W*64)

Graphics Function: *Returns the status of the most recently used graphics routine.*

Module

USE IFQWIN

Syntax

```
result = GRSTATUS( )
```

Results

The result type is INTEGER(2). The result is the status of the most recently used graphics function.

Use GRSTATUS immediately following a call to a graphics routine to determine if errors or warnings were generated. Return values less than 0 are errors, and values greater than 0 are warnings.

The following symbolic constants are defined in the `IFQWIN.F90` module file for use with GRSTATUS:

Constant	Meaning
$GRFILEWRITEERROR	Error writing bitmap file
$GRFILEOPENERROR	Error opening bitmap file
$GRIMAGEREADERROR	Error reading image
$GRBITMAPDISPLAYERROR	Error displaying bitmap
$GRBITMAPTOOLARGE	Bitmap too large
$GRIMPROPERBITMAPFORMAT	Improper format for bitmap file
$GRFILEREADERROR	Error reading file
$GRNOBITMAPFILE	No bitmap file
$GRINVALIDIMAGEBUFFER	Image buffer data inconsistent

Constant	Meaning
$GRINSUFFICIENTMEMORY	Not enough memory to allocate buffer or to complete a fill operation
$GRINVALIDPARAMETER	One or more parameters invalid
$GRMODENOTSUPPORTED	Requested video mode not supported
$GRERROR	Graphics error
$GROK	Success
$GRNOOUTPUT	No action taken
$GRCLIPPED	Output was clipped to viewport
$GRPARAMETERALTERED	One or more input parameters was altered to be within range, or pairs of parameters were interchanged to be in the proper order

After a graphics call, compare the return value of GRSTATUS to $GROK. to determine if an error has occurred. For example:

```
IF ( GRSTATUS .LT. $GROK ) THEN
! Code to handle graphics error goes here
ENDIF
```

The following routines cannot give errors, and they all set GRSTATUS to $GROK:

DISPLAYCURSOR	GETCOLORRGB	GETTEXTWINDOW
GETBKCOLOR	GETTEXTCOLOR	OUTTEXT
GETBKCOLORRGB	GETTEXTCOLORRGB	WRAPON
GETCOLOR	GETTEXTPOSITION	

The following table lists some other routines with the error or warning messages they produce for GRSTATUS:

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
ARC, ARC_W	$GRINVALIDPARAMETER	$GRNOOUTPUT
CLEARSCREEN	$GRINVALIDPARAMETER	
ELLIPSE, ELLIPSE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
FLOODFILLRGB	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT
GETARCINFO	$GRERROR	
GETFILLMASK	$GRERROR, $GRINVALIDPARAMETER	
GETFONTINFO	$GRERROR	
GETGTEXTEXTENT	$GRERROR	
GETIMAGE	$GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
GETPIXEL	$GRBITMAPTOOLARGE	
GETPIXELRGB	$GRBITMAPTOOLARGE	
LINETO, LINETO_W		$GRNOOUTPUT, $GRCLIPPED
LOADIMAGE	$GRFILEOPENERROR, $GRNOBITMAPFILE, $GRALEREADERROR, $GRIMPROPERBITMAPFORMAT, $GRBITMAPTOOLARGE, $GRIMAGEREADERROR	
OUTGTEXT		$GRNOOUTPUT, $GRCLIPPED
PIE, PIE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
POLYGON, POLYGON_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED
PUTIMAGE, PUTIMAGE_W	$GRERROR, $GRINVALIDPARAMETER, $GRINVALIDIMAGEBUFFER $GRBITMAPDISPLAYERROR	$GRPARAMETERALTERED, $GRNOOUTPUT
RECTANGLE, RECTANGLE_W	$GRINVALIDPARAMETER, $GRINSUFFICIENTMEMORY	$GRNOOUTPUT, $GRCLIPPED
REMAPPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
REMAPALLPALETTERGB	$GRERROR, $GRINVALIDPARAMETER	
SAVEIMAGE	$GRFILEOPENERROR	
SCROLLTEXTWINDOW		$GRNOOUTPUT
SETBKCOLOR	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETBKCOLORRGB	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETCLIPRGN		$GRPARAMETERALTERED
SETCOLOR		$GRPARAMETERALTERED
SETFONT	$GRERROR, $GRINSUFFICIENTMEMORY	$GRPARAMETERALTERED
SETPIXEL, SETPIXEL_W		$GRNOOUTPUT
SETPIXELRGB, SETPIXELRGB_W		$GRNOOUTPUT
SETTEXTCOLOR		$GRPARAMETERALTERED

Function	Possible GRSTATUS error codes	Possible GRSTATUS warning codes
SETTEXTCOLORRGB		$GRPARAMETERALTERED
SETTEXTPOSITION		$GRPARAMETERALTERED
SETTEXTWINDOW		$GRPARAMETERALTERED
SETVIEWPORT		$GRPARAMETERALTERED
SETWINDOW	$GRINVALIDPARAMETER	$GRPARAMETERALTERED
SETWRITEMODE	$GRINVALIDPARAMETER	

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- G
- ARC
- ELLIPSE
- FLOODFILLRGB
- LINETO
- PIE
- POLYGON
- REMAPALLPALETTERGB
- SETBKCOLORRGB
- SETCOLORRGB
- SETPIXELRGB
- SETTEXTCOLORRGB
- SETWINDOW
- SETWRITEMODE

Building Applications: Setting the Font and Displaying Text

H to I

HOSTNAM

Portability Function: *Returns the current host computer name. This function can also be specified as HOSTNM.*

Module

USE IFPORT

Syntax

`result = HOSTNAM (name)`

name	(Output) Character*(*). Name of the current host. Should be at least as long as MAX_HOSTNAM_LENGTH + 1. MAX_HOSTNAM_LENGTH is defined in the IFPORT module.

Results

The result type is INTEGER(4). The result is zero if successful. If *name* is not long enough to contain all of the host name, the function truncates the host name and returns -1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT

character(MAX_HOSTNAM_LENGTH + 1) hostnam

integer(4) istat

ISTAT = HOSTNAM (hostname)
```

HUGE

Inquiry Intrinsic Function (Generic): *Returns the largest number in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = HUGE (x)
```

x	(Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of the same type and kind parameters as *x*. If *x* is of type integer, the result has the value $r^q - 1$. If *x* is of type real, the result has the value $(1 - b^{-p})b^{e_{max}}$.

Integer parameters *r* and *q* are defined in Model for Integer Data; real parameters *b*, *p*, and e_{max} are defined in Model for Real Data.

Example

If X is of type REAL(4), HUGE (X) has the value $(1 - 2^{-24}) \times 2^{128}$.

See Also

- H to I
- TINY
- Data Representation Models

IACHAR

Elemental Intrinsic Function (Generic): *Returns the position of a character in the ASCII character set, even if the processor's default character set is different. In Intel® Fortran, IACHAR is equivalent to the ICHAR function.*

Syntax

```
result = IACHAR (c [, kind])
```

c	(Input) Must be of type character of length 1.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by kind; otherwise, the kind parameter of the result is that of default integer. If *c* is in the ASCII collating sequence, the result is the position of *c* in that sequence and satisfies the inequality (0 .le. IACHAR(c) .le. 127).

The results must be consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is true, IACHAR(C) .LE. IACHAR(D) is also true.

Example

IACHAR ('Y') has the value 89.

IACHAR ('%') has the value 37.

See Also

- H to I
- ASCII and Key Code Charts
- ACHAR
- CHAR
- ICHAR
- LGE
- LGT
- LLE
- LLT

IAND

Elemental Intrinsic Function (Generic):
Performs a logical AND on corresponding bits. This function can also be specified as AND.

Syntax

```
result = IAND (i,j)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).

j	(Input) Must be of type integer or logical with the same kind parameter as *i*. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result type is the same as *i*. The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

i	j	IAND (i, j)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIAND	INTEGER(1)	INTEGER(1)
IIAND [1]	INTEGER(2)	INTEGER(2)
JIAND	INTEGER(4)	INTEGER(4)
KIAND	INTEGER(8)	INTEGER(8)

[1] Or HIAND.

Example

IAND (2, 3) has the value 2.

IAND (4, 6) has the value 4.

See Also

- H to I
- IEOR
- IOR

- NOT

IARGC

Inquiry Intrinsic Function (Specific): *Returns the index of the last command-line argument. It cannot be passed as an actual argument. This function can also be specified as IARG or NUMARG.*

Syntax

```
result = IARGC( )
```

Results

The result type is INTEGER(4). The result is the index of the last command-line argument, which is also the number of arguments on the command line. The command is not included in the count. For example, IARGC returns 3 for the command-line invocation of `PROG1 -g -c -a`.

IARGC returns a value that is 1 less than that returned by NARGS.

Example

```
integer(4) no_of_arguments

no_of_arguments = IARGC ()

print *, 'total command line arguments are ', no_of_arguments
```

For a command-line invocation of `PROG1 -g -c -a`, the program above prints:

```
total command line arguments are 3
```

See Also

- H to I
- GETARG
- NARGS
- COMMAND_ARGUMENT_COUNT
- GET_COMMAND
- GET_COMMAND_ARGUMENT

IBCHNG

Elemental Intrinsic Function (Generic):
Reverses the value of a specified bit in an integer.

Syntax

```
result = IBCHNG (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer). This argument contains the bit to be reversed.
pos	(Input) Must be of type integer. This argument is the position of the bit to be changed. The rightmost (least significant) bit of *i* is in position 0.

Results

The result type is the same as *i*. The result is equal to *i* with the bit in position *pos* reversed.

For more information, see Bit Functions.

Example

```
INTEGER J, K

J = IBCHNG(10, 2)     ! returns 14 = 1110

K = IBCHNG(10, 1)     ! returns 8 = 1000
```

See Also

- H to I
- BTEST
- IAND
- IBCLR
- IBSET
- IEOR
- IOR
- ISHA
- ISHC
- ISHL
- ISHFT
- NOT

IBCLR

Elemental Intrinsic Function (Generic): *Clears one bit to zero.*

Syntax

```
result = IBCLR (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.

Results

The result type is the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to zero.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBCLR	INTEGER(1)	INTEGER(1)
IIBCLR[1]	INTEGER(2)	INTEGER(2)
JIBCLR	INTEGER(4)	INTEGER(4)
KIBCLR	INTEGER(8)	INTEGER(8)

[1]Or HBCLR.

Example

IBCLR (18, 1) has the value 16.

If V has the value (1, 2, 3, 4), the value of IBCLR (POS = V, I = 15) is (13, 11, 7, 15).

The following shows another example:

```
INTEGER J, K

J = IBCLR(7, 1) ! returns 5 = 0101

K = IBCLR(5, 1) ! returns 5 = 0101
```

See Also

- H to I
- BTEST
- IAND
- IBCHNG
- IBSET
- IEOR
- IOR
- ISHA
- ISHC
- ISHL
- ISHFT
- NOT

IBITS

Elemental Intrinsic Function (Generic):
Extracts a sequence of bits (a bit field).

Syntax

`result = IBITS (i,pos,len)`

i	(Input) Must be of type integer.
pos	(Input) Must be of type integer. It must not be negative and *pos*+*len* must be less than or equal to BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.
len	(Input) Must be of type integer. It must not be negative.

Results

The result type is the same as *i*. The result has the value of the sequence of *len* bits in *i*, beginning at *pos*, right-adjusted and with all other bits zero.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBITS	INTEGER(1)	INTEGER(1)
IIBITS[1]	INTEGER(2)	INTEGER(2)
JIBITS	INTEGER(4)	INTEGER(4)
KIBITS	INTEGER(8)	INTEGER(8)

[1] Or HBITS

Example

IBITS (12, 1, 4) has the value 6.

IBITS (10, 1, 7) has the value 5.

See Also

- H to I
- BTEST
- BIT_SIZE
- IBCLR
- IBSET
- ISHFT
- ISHFTC
- MVBITS

IBSET

Elemental Intrinsic Function (Generic): *Sets one bit to 1.*

Syntax

```
result = IBSET (i,pos)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
pos	(Input) Must be of type integer. It must not be negative and it must be less than BIT_SIZE(*i*). The rightmost (least significant) bit of *i* is in position 0.

Results

The result type is the same as *i*. The result has the value of the sequence of bits of *i*, except that bit *pos* of *i* is set to 1.

For more information, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BBSET	INTEGER(1)	INTEGER(1)
IIBSET[1]	INTEGER(2)	INTEGER(2)
JIBSET	INTEGER(4)	INTEGER(4)
KIBSET	INTEGER(8)	INTEGER(8)

[1]Or HBSET.

Example

IBSET (8, 1) has the value 10.

If V has the value (1, 2, 3, 4), the value of IBSET (POS = V, I = 2) is (2, 6, 10, 18).

The following shows another example:

```
INTEGER I

I = IBSET(8, 2) ! returns 12 = 1100
```

See Also

- H to I
- BTEST
- IAND

- IBCHNG
- IBCLR
- IEOR
- IOR
- ISHA
- ISHC
- ISHL
- ISHFT
- NOT

ICHAR

Elemental Intrinsic Function (Generic):
Returns the position of a character in the processor's character set.

Syntax

```
result = ICHAR (c [, kind])
```

c	(Input) Must be of type character of length 1.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer.

The result value is the position of *c* in the processor's character set. *c* is in the range zero to n - 1, where *n* is the number of characters in the character set.

For any characters C and D (capable of representation in the processor), C .LE. D is true only if ICHAR(C) .LE. ICHAR(D) is true, and C .EQ. D is true only if ICHAR(C) .EQ. ICHAR(D) is true.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(2)
ICHAR [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

Specific Name	Argument Type	Result Type

[1]This specific function cannot be passed as an actual argument.

Example

ICHAR ('W') has the value 87.

ICHAR ('#') has the value 35.

See Also

- H to I
- IACHAR
- CHAR
- ASCII and Key Code Charts

IDATE Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns three integer values representing the current month, day, and year. IDATE can be used as an intrinsic subroutine or as a portability subroutine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL IDATE (i, j, k)
```

i	(Output) Must be of type integer. It is the current month.
j	(Output) Must be of type integer with the same kind type parameter as *i*. It is the current day.
k	(Output) Must be of type integer with the same kind type parameter as *i*. It is the current year.

The current month is returned in *i*; the current day in *j*. The last two digits of the current year are returned in *k*.

CAUTION. The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Example

If the current date is September 16, 1999, the values of the integer variables upon return are: I = 9, J = 16, and K = 99.

See Also

- H to I
- DATE intrinsic procedure
- DATE_AND_TIME
- GETDAT
- IDATE portability routine

IDATE Portability Routine

Portability Subroutine: *Returns the month, day, and year of the current system. IDATE can be used as an intrinsic subroutine or as a portability subroutine. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

```
CALL IDATE (i, j, k)
```

-or-

```
CALL IDATE (iarray)
```

i	(Output) INTEGER(4). Is the current system month.
j	(Output) INTEGER(4). Is the current system day.
k	(Output) INTEGER(4). Is the current system year as an offset from 1900.
iarray	(Output) INTEGER(4). Is a three-element array that holds day as element 1, month as element 2, and year as element 3. The month is between 1 and 12. The year is greater than or equal to 1969 and is returned as 2 digits.

CAUTION. The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIME instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
    use IFPORT
    integer(4) imonth, iday, iyear, datarray(3)
! If the date is July 11, 1999:
    CALL IDATE(IMONTH, IDAY, IYEAR)
! sets IMONTH to 7, IDAY to 11 and IYEAR to 99.
    CALL IDATE (DATARRAY)
! datarray is (/11,7,99/)
```

See Also

- H to I
- DATE portability routine
- DATE_AND_TIME
- GETDAT
- IDATE intrinsic procedure

IDATE4

Portability Subroutine: *Returns the month, day, and year of the current system.*

Module

USE IFPORT

Syntax

`CALL IDATE4 (i,j,k)`

-or-

`CALL IDATE4 (iarray)`

i	(Output) INTEGER(4). The current system month.
j	(Output) INTEGER(4). The current system day.
k	(Output) INTEGER(4). The current system year as an offset from 1900.
iarray	(Output) INTEGER(4). A three-element array that holds day as element 1, month as element 2, and year as element 3. The month is between 1 and 12. The year is returned as an offset from 1900, if the year is less than 2000. For years greater than or equal to 2000, this element simply returns the integer year, such as 2003.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

IDENT

General Compiler Directive: *Specifies a string that identifies an object module. The compiler places the string in the identification field of an object module when it generates the module for each source program unit.*

Syntax

*c*DEC$ IDENT *string*

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
string	Is a character constant containing printable characters. The number of characters is limited by the length of the source line.

Only the first IDENT directive is effective; the compiler ignores any additional IDENT directives in a program unit or module.

IDFLOAT

Portability Function: *Converts an INTEGER(4) argument to double-precision real type.*

Module

USE IFPORT

Syntax

```
result = IDFLOAT (i)
```

i (Input) Must be of type INTEGER(4).

Results

The result type is double-precision real (REAL(8) or REAL*8).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- DFLOAT

IEEE_CLASS

Elemental Module Intrinsic Function (Generic): *Returns the IEEE class.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_CLASS (x)
```

x (Input) Must be of type REAL.

Results

The result is of type TYPE(IEEE_CLASS_TYPE). The result value is one of the following:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUITE_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO

IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO

IEEE_CLASS does not return IEEE_OTHER_VALUE in Intel Fortran.

Example

IEEE_CLASS(1.0) has the value IEEE_POSITIVE_NORMAL.

IEEE_COPY_SIGN

Elemental Module Intrinsic Function (Generic): *Returns an argument with a copied sign. This is equivalent to the IEEE copysign function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_COPY_SIGN (x,y)`

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type is the same as x. The result has the value x with the sign of y. This is true even for IEEE special values, such as NaN or infinity.

The flags information is returned as a set of 1-bit flags.

Example

The value of IEEE_COPY_SIGN (X,3.0) is ABS (X), even when X is NaN.

IEEE_GET_FLAG

Elemental Module Intrinsic Subroutine (Generic): *Returns whether an exception flag is signaling.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_GET_FLAG (*flag*, *flag_value*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
flag_value	(Output) Must be of type default logical. If the exception in 'flag' is signaling, the result is true; otherwise, false.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS ! Can also use module IEEE_ARITHMETIC

LOGICAL ON

...

CALL IEEE_GET_FLAG(IEEE_INVALID, ON)
```

If flag IEEE_INVALID is signaling, the value of ON is true; if it is quiet, the value of ON is false.

IEEE_GET_HALTING_MODE

Elemental Module Intrinsic Subroutine (Generic): *Stores the halting mode for an exception.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_GET_HALTING_MODE (*flag*, *halting*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
halting	(Output) Must be of type default logical. If the exception in "flag" causes halting, the result is true; otherwise, false.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS

LOGICAL HALT

...

CALL IEEE_GET_HALTING_MODE(IEEE_INVALID, HALT) ! Stores the halting mode

CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.) ! Stops halting

...

CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, HALT) ! Restores halting
```

IEEE_GET_ROUNDING_MODE

Intrinsic Module Subroutine (Generic): *Stores the current IEEE rounding mode.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_GET_ROUNDING_MODE (*round_value*)

round_value	(Output) Must be scalar and of type TYPE (IEEE_ROUND_TYPE). It returns one of the following IEEE floating-point rounding values: IEEE_DOWN, IEEE_NEAREST, IEEE_TO_ZERO, or IEEE_UP; otherwise, IEEE_OTHER. The result can only be used if IEEE_SET_ROUNDING_MODE is invoked.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND
...
CALL IEEE_GET_ROUNDING_MODE(ROUND) ! Stores the rounding mode
CALL IEEE_SET_ROUNDING_MODE(IEEE_UP) ! Resets the rounding mode
...
CALL IEEE_SET_ROUNDING_MODE(VALUE) ! Restores the previous rounding mode
```

IEEE_GET_STATUS

Intrinsic Module Subroutine (Generic): *Stores the current state of the floating-point environment.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_GET_STATUS (*status_value*)

status_value	(Input) Must be scalar and of type TYPE (IEEE_STATUS_TYPE). It stores the floating-point status. The result can only be used if IEEE_SET_STATUS is invoked.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS !Can also use IEEE_ARITHMETIC
TYPE(IEEE_ STATUS_TYPE) STATUS
...
CALL IEEE_GET_STATUS(STATUS) ! Stores the floating-point status
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Sets all flags to be quiet
...
CALL IEEE_SET_STATUS(STATUS) ! Restores the floating-point status
```

IEEE_GET_UNDERFLOW_MODE

Intrinsic Module Subroutine (Generic): *Stores the current underflow mode.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

```
CALL IEEE_GET_UNDERFLOW_MODE (gradual)
```

gradual	(Output) Must be default logical scalar. The result is true if the current underflow mode is gradual (IEEE denormals are allowed) and false if the current underflow mode is abrupt (underflowed results are set to zero).

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS

LOGICAL GRAD
...
CALL IEEE_GET_UNDERFLOW_MODE(GRAD)
IF (GRAD) THEN ! underflows are gradual
...
ELSE ! underflows are abrupt
...
END IF
```

IEEE_IS_FINITE

Elemental Module Intrinsic Function (Generic): *Returns whether an IEEE value is finite.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_FINITE (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of x is finite; otherwise, false.

An IEEE value is finite if IEEE_CLASS(x) has one of the following values:

IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_NORMAL	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_DENORMAL	IEEE_NEGATIVE_ZERO

Example

IEEE_IS_FINITE (-2.0) has the value true.

IEEE_IS_NAN

Elemental Module Intrinsic Function (Generic): *Returns whether an IEEE value is Not-a-Number (NaN).*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_NAN (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of x is NaN; otherwise, false.

Example

IEEE_IS_NAN (SQRT(-2.0)) has the value true if IEEE_SUPPORT_SQRT (2.0) has the value true.

IEEE_IS_NEGATIVE

Elemental Module Intrinsic Function (Generic): *Returns whether an IEEE value is negative.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_IS_NEGATIVE (x)`

x	(Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of x is negative; otherwise, false.

An IEEE value is negative if IEEE_CLASS(x) has one of the following values::

IEEE_NEGATIVE_NORMAL	IEEE_NEGATIVE_ZERO
IEEE_NEGATIVE_DENORMAL	IEEE_NEGATIVE_INF

Example

IEEE_IS_NEGATIVE (2.0) has the value false.

IEEE_IS_NORMAL

Elemental Module Intrinsic Function (Generic): *Returns whether an IEEE value is normal.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_IS_NORMAL (x)
```

x (Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if the value of x is normal; otherwise, false.

An IEEE value is normal if IEEE_CLASS(x) has one of the following values:

IEEE_POSITIVE_NORMAL	IEEE_POSITIVE_ZERO
IEEE_NEGATIVE_NORMAL	IEEE_NEGATIVE_ZERO

Example

IEEE_IS_NORMAL (SQRT(-2.0)) has the value false if IEEE_SUPPORT_SQRT (-2.0) has the value true.

IEEE_LOGB

Elemental Module Intrinsic Function (Generic): *Returns a floating-point value equal to the unbiased exponent of the argument. This is equivalent to the IEEE logb function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_LOGB (x)
```

x (Input) Must be of type REAL.

Results

The result type is the same as x. The result has the value of the unbiased exponent of x if the value of x is not zero, infinity, or NaN. The value of the result is equal to EXPONENT(x) - 1.

If x is equal to 0, the result is -infinity if IEEE_SUPPORT_INF(x) is true; otherwise, -HUGE(x). In either case, the IEEE_DIVIDE_BY_ZERO exception is signaled.

Example

IEEE_LOGB (3.4) has the value 1.0; IEEE_LOGB (4.0) has the value 2.0.

IEEE_NEXT_AFTER

Elemental Module Intrinsic Function (Generic): *Returns the next representable value after X toward Y. This is equivalent to the IEEE nextafter function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_NEXT_AFTER (x,y)`

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type is the same as x. If x is equal to y, the result is x; no exception is signaled. If x is not equal to y, the result has the value of the next representable neighbor of x toward y. The neighbors of zero (of either sign) are both nonzero.

The following exceptions are signaled under certain cases:

Exception	Signalled
IEEE_OVERFLOW	When X is finite but IEEE_NEXT_AFTER(X,Y) is infinite
IEEE_UNDERFLOW	When IEEE_NEXT_AFTER(X,Y) is denormalized
IEEE_INEXACT	In both the above cases

Example

The value of IEEE_NEXT_AFTER (2.0,3.0) is 2.0 + EPSILON (X).

IEEE_REM

Elemental Module Intrinsic Function (Generic): *Returns the result value from a remainder operation. This is equivalent to the IEEE rem function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_REM (x,y)`

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type is real with the kind type parameter of whichever argument has greater precision.

Regardless of the rounding mode, the result value is x - y*N, where N is the integer nearest to the value x / y. If |N – x / y | = 1/2, N is even. If the result value is zero, the sign is the same as x.

Example

The value of IEEE_REM (5.0,4.0) is 1.0; the value of IEEE_REM (2.0,1.0) is 0.0; the value of IEEE_REM (3.0,2.0) is -1.0.

IEEE_RINT

Elemental Module Intrinsic Function (Generic): *Returns an integer value rounded according to the current rounding mode.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_RINT (x)`

x	(Input) Must be of type REAL.

Results

The result type is the same as x. The value of the result is x rounded to an integer according to the current rounding mode. If the result value is zero, the sign is the same as x.

Example

If the current rounding mode is IEEE_UP, the value of IEEE_RINT (2.2) is 3.0.

If the current rounding mode is IEEE_NEAREST, the value of IEEE_RINT (2.2) is 2.0.

IEEE_SCALB

Elemental Module Intrinsic Function (Generic): *Returns the exponent of a radix-independent floating-point number. This is equivalent to the IEEE scalb function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SCALB (x,i)
```

x	(Input) Must be of type REAL.
i	(Input) Must be of type INTEGER.

Results

The result type is the same as x. The result is x multiplied by 2**i, if the value can be represented as a normal number.

If x (2**i) is too small and there is a loss of accuracy, the exception IEEE_UNDERFLOW is signaled. The result value is the nearest number that can be represented with the same sign as x.

If x is finite and x (2**i) is too large, an IEEE_OVERFLOW exception occurs. If IEEE_SUPPORT_INF (x) is true, the result value is infinity with the same sign as x; otherwise, the result value is SIGN (HUGE(x), x).

If x is infinite, the result is the same as x; no exception is signaled.

Example

The value of IEEE_SCALB (2.0,3) is 16.0.

IEEE_SELECTED_REAL_KIND

Transformational Module Intrinsic Function (Generic): *Returns the the value of the kind parameter of an IEEE REAL data type.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SELECTED_REAL_KIND ([p][,r])`

p	(Input; optional) Must be scalar and of type INTEGER.
r	(Input; optional) Must be scalar and of type INTEGER.

At least one argument must be specified.

Results

The result is a scalar of type default integer. The result has a value equal to a value of the kind parameter of an IEEE real data type with decimal precision, as returned by the function PRECISION, of at least `p` digits and a decimal exponent range, as returned by the function RANGE, of at least `r`.

If no such kind type parameter is available on the processor, the result is as follows:

- -1 if the precision is not available
- -2 if the exponent range is not available
- -3 if neither the precision nor the exponent range is available
- -4 if one but not both of the precision and the exponent range is available.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision.

Example

IEEE_SELECTED_REAL_KIND (6, 70) = 8.

See Also

- H to I
- Model for Real Data

IEEE_SET_FLAG

Elemental Module Intrinsic Function (Generic): *Assigns a value to an exception flag.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_FLAG (*flag*,*flag_value*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
flag_value	(Output) Must be of type default logical. If it has the value true, the exception in 'flag' is set to signal; otherwise, the exception is set to be quiet.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS ! Can also use module IEEE_ARITHMETIC

...

CALL IEEE_SET_FLAG (IEEE_INVALID, .TRUE.) ! Sets the IEEE_INVALID flag to signal
```

IEEE_SET_HALTING_MODE

Elemental Module Intrinsic Function (Generic): *Controls halting or continuation after an exception.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_HALTING_MODE (*flag*, *halting*)

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW
halting	(Input) Must be scalar and of type default logical. If the value is true, the exception specified in FLAG will cause halting; otherwise, execution will continue after this exception.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS

LOGICAL HALT

...

CALL IEEE_GET_HALTING_MODE(IEEE_INVALID, HALT) ! Stores the halting mode

CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.) ! Stops halting

...

CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, HALT) ! Restores halting
```

IEEE_SET_ROUNDING_MODE

Intrinsic Module Subroutine (Generic): *Sets the IEEE rounding mode.*

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_ROUNDING_MODE (*round_value*)

round_value	(Output) Must be scalar and of type TYPE (IEEE_ROUND_TYPE). It specifies one of the following IEEE floating-point rounding values: IEEE_DOWN, IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, or IEEE_OTHER.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_ARITHMETIC

TYPE (IEEE_ROUND_TYPE) ROUND
...
CALL IEEE_GET_ROUNDING_MODE (ROUND) ! Stores the rounding mode
CALL IEEE_SET_ROUNDING_MODE (IEEE_UP) ! Resets the rounding mode
...
CALL IEEE_SET_ROUNDING_MODE (VALUE) ! Restores the previous rounding mode
```

IEEE_SET_STATUS

Intrinsic Module Subroutine (Generic):
Restores the state of the floating-point environment.

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

CALL IEEE_SET_STATUS (*status_value*)

status_value	(Input) Must be scalar and of type TYPE (IEEE_STATUS_TYPE). Its value must be set in a previous invocation of IEEE_GET_STATUS.

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS !Can also use IEEE_ARITHMETIC
TYPE (IEEE_STATUS_TYPE) STATUS
...
CALL IEEE_GET_STATUS (STATUS) ! Stores the floating-point status
CALL IEEE_SET_FLAG (IEEE_ALL,.FALSE.) ! Sets all flags to be quiet
...
CALL IEEE_SET_STATUS (STATUS) ! Restores the floating-point status
```

IEEE_SET_UNDERFLOW_MODE

Intrinsic Module Subroutine (Generic): *Sets the current underflow mode.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

CALL IEEE_SET_UNDERFLOW_MODE (*gradual*)

gradual	(Input) Must be scalar and of type default logical. If it is true, the current underflow mode is set to gradual underflow (denormals may be produced on underflow). If it is false, the current underflow mode is set to abrupt (underflowed results are set to zero).

Example

Consider the following:

```
USE, INTRINSIC :: IEEE_EXCEPTIONS
```

```
LOGICAL :: SG
...
CALL IEEE_GET_UNDERFLOW_MODE (SG) ! Stores underflow mode
CALL IEEE_SET_UNDERFLOW_MODE (.FALSE.) ! Resets underflow mode
...                                    ! Abrupt underflows happens here
CALL IEEE_SET_UNDERFLOW_MODE (SG) ! Restores previous undeflow mode
```

IEEE_SUPPORT_DATATYPE

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE arithmetic.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_SUPPORT_DATATYPE  ([x])`

x (Input; optional) Must be scalar and of type REAL.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports IEEE arithmetic for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports IEEE arithmetic for real variables of the same kind type parameter as x; otherwise, false.

If real values are implemented according to the IEEE standard except that underflowed values flush to zero (abrupt) instead of being denormal.

Example

IEEE_SUPPORT_DATATYPE (3.0) has the value true.

IEEE_SUPPORT_DENORMAL

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE denormalized numbers.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_DENORMAL  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports arithmetic operations and assignments with denormalized numbers for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports arithmetic operations and assignments with denormalized numbers for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_DENORMAL () has the value true if IEEE denormalized numbers are supported for all real types.

IEEE_SUPPORT_DIVIDE

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE divide.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_DIVIDE  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports divide with the accuracy specified by the IEEE standard for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports divide with the accuracy specified by the IEEE standard for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_DIVIDE ( ) has the value true if IEEE divide is supported for all real types.

IEEE_SUPPORT_FLAG

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE exceptions.

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

```
result = IEEE_SUPPORT_FLAG  (flag [, x])
```

flag	(Input) Must be a scalar of type TYPE (IEEE_FLAG_TYPE). Its value is one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.
x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports detection of the exception specified by "flag" for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports detection of the exception specified by "flag" for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_FLAG (IEEE_UNDERFLOW) has the value true if the IEEE_UNDERFLOW exception is supported for all real types.

IEEE_SUPPORT_HALTING

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE halting.

Module

USE, INTRINSIC :: IEEE_EXCEPTIONS

Syntax

```
result = IEEE_SUPPORT_HALTING(flag)
```

flag	(Input) Must be of type TYPE (IEEE_FLAG_TYPE). It specifies one of the following IEEE flags: IEEE_DIVIDE_BY_ZERO, IEEE_INEXACT, IEEE_INVALID, IEEE_OVERFLOW, or IEEE_UNDERFLOW.

Results

The result is a scalar of type default logical. The result has the value true if the processor supports the ability to control halting after the exception specified by "flag"; otherwise, false.

Example

IEEE_SUPPORT_HALTING (IEEE_UNDERFLOW) has the value true if halting is supported after an IEEE_UNDERFLOW exception

IEEE_SUPPORT_INF

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE infinities.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_INF  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports IEEE infinities (positive and negative) for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports IEEE infinities for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_INF( ) has the value true if IEEE infinities are supported for all real types.

IEEE_SUPPORT_IO

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE base conversion rounding during formatted I/O.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_IO  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports base conversion rounding during formatted input and output for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports base conversion rounding during formatted input and output for real variables of the same kind type parameter as x; otherwise, false.

The base conversion rounding applies to modes IEEE_UP, IEEE_DOWN, IEEE_TO_ZERO, and IEEE_NEAREST.

Example

IEEE_SUPPORT_IO ( ) has the value true if base conversion rounding is supported for all real types during formatted I/O.

IEEE_SUPPORT_NAN

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE Not-a-Number feature.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_NAN  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports NaNs for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports NaNs for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_NAN () has the value true if IEEE NaNs are supported for all real types.

IEEE_SUPPORT_ROUNDING

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE rounding mode.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_ROUNDING  (round_value [, x])
```

round_value	(Input) Must be of type TYPE(IEEE_ROUND_TYPE). It specifies one of the following rounding modes: IEEE_UP, IEEE_DOWN, IEEE_TO_ZERO, and IEEE_NEAREST.
x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports the rounding mode specified by round_value for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports the rounding mode specified by round_value for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_ROUNDING (IEEE_DOWN) has the value true if rounding mode IEEE_DOWN is supported for all real types.

IEEE_SUPPORT_SQRT

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE SQRT (square root).

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_SQRT  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor implements SQRT in accord with the IEEE standard for all real values; otherwise, false.

If x is specified, the result has the value true if the processor implements SQRT in accord with the IEEE standard for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_SQRT ( ) has the value true if IEEE SQRT is supported for all real types.

IEEE_SUPPORT_STANDARD

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports IEEE features defined in the standard.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_STANDARD  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. The result has the value true if the results of all the following functions are true (x can be omitted):

IEEE_SUPPORT_DATATYPE([x])

IEEE_SUPPORT_DENORMAL([x])

IEEE_SUPPORT_DIVIDE([x])

IEEE_SUPPORT_FLAG(flag [, x])[1]

IEEE_SUPPORT_HALTING(flag)[1]

IEEE_SUPPORT_INF([x])

IEEE_SUPPORT_NAN([x])

IEEE_SUPPORT_ROUNDING(round_value [, x])[2]

IEEE_SUPPORT_SQRT([x])

1: "flag" must be a valid value

2: "round_value" must be a valid value

Otherwise, the result has the value, false.

Example

IEEE_SUPPORT_STANDARD () has the value false if both IEEE and non-IEEE real kinds are supported.

IEEE_SUPPORT_UNDERFLOW_CONTROL

Inquiry Module Intrinsic Function (Generic):
Returns whether the processor supports the ability to control the underflow mode.

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

```
result = IEEE_SUPPORT_UNDERFLOW_CONTROL  ([x])
```

x	(Input; optional) Must be of type REAL; it can be scalar or array valued.

Results

The result is a scalar of type default logical. If x is omitted, the result has the value true if the processor supports controlling the underflow mode for all real values; otherwise, false.

If x is specified, the result has the value true if the processor supports controlling the underflow mode for real variables of the same kind type parameter as x; otherwise, false.

Example

IEEE_SUPPORT_UNDERFLOW _CONTROL () has the value true if controlling the underflow mode is supported for all real types.

IEEE_UNORDERED

Elemental Module Intrinsic Function (Generic): *Returns whether one or more of the arguments is Not-a-Number (NaN). This is equivalent to the IEEE unordered function.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_UNORDERED (x,y)`

x	(Input) Must be of type REAL.
y	(Input) Must be of type REAL.

Results

The result type is default logical. The result has the value true if x or y is a NaN, or both are NaNs; otherwise, false.

Example

IEEE_UNORDERED (0.0, SQRT(-2.0)) has the value true if IEEE_SUPPORT_SQRT (2.0) has the value true.

IEEE_VALUE

Elemental Module Intrinsic Function (Generic): *Creates an IEEE value.*

Module

USE, INTRINSIC :: IEEE_ARITHMETIC

Syntax

`result = IEEE_VALUE (x, class)`

x	(Input) Must be of type REAL.
class	(Input) Must be of type TYPE (IEEE_CLASS_TYPE). Its value is one of the following:

IEEE_SIGNALING_NAN	IEEE_NEGATIVE_NORMAL
IEEE_QUITE_NAN	IEEE_POSITIVE_DENORMAL
IEEE_POSITIVE_INF	IEEE_NEGATIVE_DENORMAL
IEEE_NEGATIVE_INF	IEEE_POSITIVE_ZERO
IEEE_POSITIVE_NORMAL	IEEE_NEGATIVE_ZERO

Results

The result type is the same as x. The result value is an IEEE value as specified by "class".

When IEEE_VALUE returns a signaling NaN, it is processor dependent whether or not invalid is signaled and processor dependent whether or not the signaling NaN is converted to a quiet NaN.

Example

IEEE_VALUE (1.0,IEEE_POSITIVE_INF) has the value +infinity.

IEEE_FLAGS

Portability Function: *Gets, sets or clears IEEE flags for rounding direction and precision as well as queries or controls exception status. This function provides easy access to the modes and status required to use the features of IEEE Standard 754-1985 arithmetic in a Fortran program.*

Module

USE IFPORT

Syntax

```
result = IEEE_FLAGS (action,mode,in,out)
```

action	(Input) Character*(*). One of the following literal values: 'GET', 'SET', 'CLEAR', or 'CLEARALL'.

mode (Input) Character*(*). One of the following literal values: 'direction', 'precision', or 'exception'. The value 'precision' is only allowed on Intel® 64 architecture and IA-32 architecture.

in (Input) Character*(*). One of the following literal values: 'inexact', 'division', 'underflow','overflow', 'invalid', 'all', 'common', 'nearest', 'tozero', 'negative', 'positive','extended', 'double', 'single', or ' ', which represents an unused (null) value.

out (Output) Must be at least CHARACTER*9. One of the literal values listed for *in*.

The descriptions for the values allowed for *in* and *out* can be summarized as follows:

Value	Description
'nearest'	Rounding direction flags
'tozero'	
'negative'	
'positive'	
'single'	Rounding precision flags
'double'	
'extended'	
'inexact'	Math exception flags
'underflow'	
'overflow'	
'division'	
'invalid'	
'all'	All math exception flags above
'common'	The math exception flags: 'invalid', 'division', 'overflow', and 'underflow'

The values for *in* and *out* depend on the *action* and *mode* they are used with. The interaction of the parameters can be summarized as follows:

Value of *action*	**Value of *mode***	**Value of *in***	**Value of *out***	**Functionality and return value**
GET	'direction'	Null (' ')	One of 'nearest', 'tozero', 'negative', or 'positive'	Tests rounding direction settings. Returns the current setting, or 'not available'.
	'exception'	Null (' ')	One of 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', or 'common'	Tests math exception settings. Returns the current setting, or 0.
	'precision'	Null (' ')	One of 'single ', 'double ', or 'extended'	Tests rounding precision settings. Returns the current setting, or 'not available'.

Value of *action*	Value of *mode*	Value of *in out*	Value of *out*	Functionality and return value
SET	'direction'	One of 'nearest', 'tozero', 'negative', or 'positive'	Null (' ')	Sets a rounding direction.
	'exception'	One of 'inexact', 'division', 'underflow', 'overflow', 'invalid', 'all', or 'common'	Null (' ')	Sets a floating-point math exception.
	'precision'	One of 'single ', 'double ', or 'extended'	Null (' ')	Sets a rounding precision.
CLEAR	'direction'	Null (' ')	Null (' ')	Clears the *mode*. Sets rounding to 'nearest'. Returns 0 if successful.
	'exception'	One of 'inexact', 'division',	Null (' ')	Clears the *mode*.

Value of *action*	Value of *mode*	Value of *in*	Value of *out*	Functionality and return value
		'underflow', 'overflow', 'invalid', 'all', or 'common'		Returns 0 if successful.
	'precision'	Null (' ')	Null (' ')	Clears the *mode*. Sets precision to 'double' (W*32, W*64) or 'extended' (L*X, M*X). Returns 0 if successful.
CLEARALL	Null (' ')	Null (' ')	Null (' ')	Clears all flags. Sets rounding to 'nearest', sets precision to 'double' (W*32, W*64) or 'extended' (L*X, M*X), and sets all exception flags to 0.

Value of *action*	Value of *mode*	Value of *in*	Value of *out*	Functionality and return value
				Returns 0 if successful.

Results

IEEE_FLAGS is an elemental, integer-valued function that sets IEEE flags for GET, SET, CLEAR, or CLEARALL procedures. It lets you control rounding direction and rounding precision, query exception status, and control exception enabling or disabling by using the SET or CLEAR procedures, respectively.

The flags information is returned as a set of 1-bit flags.

Example

The following example gets the highest priority exception that has a flag raised. It passes the input argument *in* as a null string:

```
USE IFPORT

INTEGER*4 iflag

CHARACTER*9 out

iflag = ieee_flags('get', 'exception', '', out)

PRINT *, out, ' flag raised'
```

The following example sets the rounding direction to round toward zero, unless the hardware does not support directed rounding modes:

```
USE IFPORT

INTEGER*4 iflag

CHARACTER*10 mode, out, in

iflag = ieee_flags('set', 'direction', 'tozero', out)
```

The following example sets the rounding direction to the default ('nearest'):

```
USE IFPORT

INTEGER*4 iflag

CHARACTER*10 out, in

iflag = ieee_flags('clear','direction', '', '' )
```

The following example clears all exceptions:

```
USE IFPORT

INTEGER*4 iflag

CHARACTER*10 out

iflag = ieee_flags('clear','exception', 'all', '' )
```

The following example restores default direction and precision settings, and sets all exception flags to 0:

```
USE IFPORT

INTEGER*4 iflag

CHARACTER*10 mode, out, in

iflag = ieee_flags('clearall', '', '', '')
```

The following example detects an underflow exception:

```
USE IFPORT

CHARACTER*20 out, in

excep_detect = ieee_flags('get', 'exception', 'underflow', out)

if (out .eq.'underflow') stop 'underflow'
```

IEEE_HANDLER

Portability Function: *Establishes a handler for IEEE exceptions.*

Module

USE IFPORT

Syntax

`result = IEEE_HANDLER (`*`action, exception, handler`*`)`

action	(Input) Character*(*). One of the following literal IEEE actions: 'GET', 'SET', or 'CLEAR'. For more details on these actions, see IEEE_FLAGS.
exception	(Input) Character*(*). One of the following literal IEEE exception flags: 'inexact', 'underflow', 'overflow', 'division', 'invalid', 'all' (which equals the previous five flags), or 'common' (which equals 'invalid', 'overflow', 'underflow', and 'division'). The flags 'all' or 'common' should only be used for actions SET or CLEAR.
handler	(Input) The address of an external signal-handling routine.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The result is 0 if successful; otherwise, 1.

IEEE_HANDLER calls a signal-handling routine to establish a handler for IEEE exceptions. It also enables an FPU trap corresponding to the required exception.

The state of the FPU is not defined in the handler routine. When the FPU trap occurs, the program invokes the handler routine. After the handler routine is executed, the program terminates.

The handler routine gets the exception code in the SIGINFO argument. SIGNO is the number of the system signal. The meaning of the SIGINFO constants appear in the following table (defined in the IFPORT module):

FPE$INVALID	Invalid operation
FPE$ZERODIVIDE	Divide-by-zero
FPE$OVERFLOW	Numeric overflow
FPE$UNDERFLOW	Numeric underflow
FPE$INEXACT	Inexact result (precision)

'GET' actions return the location of the current handler routine for exception cast to an INTEGER.

Example

The following example creates a handler routine and sets it to trap divide-by-zero:

```
PROGRAM TEST_IEEE
 REAL :: X, Y, Z
 CALL FPE_SETUP
 X = 0.
 Y = 1.
 Z = Y / X
END PROGRAM
SUBROUTINE FPE_SETUP
USE IFPORT
IMPLICIT NONE
 INTERFACE
   SUBROUTINE FPE_HANDLER(SIGNO, SIGINFO)
     INTEGER(4), INTENT(IN) :: SIGNO, SIGINFO
   END SUBROUTINE
 END INTERFACE
 INTEGER IR
 IR = IEEE_HANDLER('set','division',FPE_HANDLER)
END SUBROUTINE FPE_SETUP
SUBROUTINE FPE_HANDLER(SIG, CODE)
 USE IFPORT
 IMPLICIT NONE
 INTEGER SIG, CODE
 IF(CODE.EQ.FPE$ZERODIVIDE) PRINT *,'Occurred divide by zero.'
 CALL ABORT
END SUBROUTINE FPE_HANDLER
```

See Also

- H to I
- IEEE_FLAGS

IEOR

Elemental Intrinsic Function (Generic):
Performs an exclusive OR on corresponding bits.
This function can also be specified as XOR or IXOR.

Syntax

```
result = IEOR (i,j)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
j	(Input) Must be of type integer with the same kind parameter as *i*. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result type is the same as *i*. The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

i	j	IEOR (i, j)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIEOR[1]	INTEGER(1)	INTEGER(1)
IIEOR[2]	INTEGER(2)	INTEGER(2)
JIEOR[3]	INTEGER(4)	INTEGER(4)

Specific Name	Argument Type	Result Type
KIEOR	INTEGER(8)	INTEGER(8)

[1]Or BIXOR

[2]Or HIEOR, HIXOR, or IIXOR

[3]Or JIXOR

Example

IEOR (12, 7) has the value 11; binary 1100 exclusive OR with binary 0111 is binary 1011.

The following shows another example:

```
INTEGER I

I = IEOR(240, 90)  ! returns 170

                   ! IEOR (B'11110000', B'1011010') == B'10101010'
```

The following shows an example using alternate option XOR:

```
 INTEGER i, j, k

 i = 3             ! B'011'

 j = 5             ! B'101'

 k = XOR(i, j)     ! returns 6 = B'110'
```

See Also

- H to I
- IAND
- IOR
- NOT

IERRNO

Portability Function: *Returns the number of the last detected error from any routines in the IFPORT module that return error codes.*

Module

USE IFPORT

Syntax

```
result = IERRNO( )
```

Results

The result type is INTEGER(4). The result value is the last error code from any portability routines that return error codes. These error codes are analogous to `errno` on a Linux* or Mac OS* X system. The module `IFPORT.F90` provides parameter definitions for the following `errno` names (typically found in `errno.h` on Linux systems):

Symbolic name	Number	Description
EPERM	1	Insufficient permission for operation
ENOENT	2	No such file or directory
ESRCH	3	No such process
EIO	5	I/O error
E2BIG	7	Argument list too long
ENOEXEC	8	File is not executable
ENOMEM	12	Not enough resources
EACCES	13	Permission denied
EXDEV	18	Cross-device link
ENOTDIR	20	Not a directory
EINVAL	22	Invalid argument

The value returned by IERRNO is updated only when an error occurs. For example, if an error occurs on a GETLOG call and then two CHMOD calls succeed, a subsequent call to IERRNO returns the error for the GETLOG call.

Examine IERRNO immediately after returning from a portability routine. Other Fortran routines, as well as any Windows* APIs, can also change the error code to an undefined value. IERRNO is set on a per thread basis.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
CHARACTER*20 username
INTEGER(4) ierrval
ierrval=0 !initialize return value
CALL GETLOG(username)
IF (IERRNO( ) == ierrval) then
  print *, 'User name is ',username
  exit
ELSE
  ierrval = ierrno()
  print *, 'Error is ',ierrval
END IF
```

IF - Arithmetic

Statement: *Conditionally transfers control to one of three statements, based on the value of an arithmetic expression. It is an obsolescent feature in Fortran 95 and Fortran 90.*

Syntax

`IF (`*`expr`*`)` *`label1`*`,`*`label2`*`,`*`label3`*

`expr`	Is a scalar numeric expression of type integer or real (enclosed in parentheses).
`label1, label2, label3`	Are the labels of valid branch target statements that are in the same scoping unit as the arithmetic **IF**statement.

Description

All three labels are required, but they do not need to refer to three different statements. The same label can appear more than once in the same arithmetic IF statement.

During execution, the expression is evaluated first. Depending on the value of the expression, control is then transferred as follows:

If the Value of *expr* is:	Control Transfers To:
Less than 0	Statement *label1*
Equal to 0	Statement *label2*
Greater than 0	Statement *label3*

Example

The following example transfers control to statement 50 if the real variable THETAis less than or equal to the real variable CHI. Control passes to statement 100 only if THETAis greater than CHI.

```
IF (THETA-CHI) 50,50,100
```

The following example transfers control to statement 40 if the value of the integer variable NUMBERis even. It transfers control to statement 20 if the value is odd.

```
IF (NUMBER / 2*2 - NUMBER) 20,40,20
```

The following statement transfers control to statement 10 for n < 10, to statement 20 for n = 10, and to statement 30 for n > 10:

```
IF (n-10) 10, 20, 30
```

The following statement transfers control to statement 10 if n <= 10, and to statement 30 for n > 10:

```
IF (n-10) 10, 10, 30
```

See Also

- H to I
- SELECT CASE...END SELECT
- Execution Control
- Obsolescent Features in Fortran 90

IF - Logical

Statement: *Conditionally executes one statement based on the value of a logical expression. (This statement was called a logical IF statement in FORTRAN 77.)*

Syntax

```
IF (expr) stmt
```

expr	Is a scalar logical expression enclosed in parentheses.
stmt	Is any complete, unlabeled, executable Fortran statement, except for the following: • A CASE, DO, IF, FORALL, or WHERE construct • Another IF statement • The END statement for a program, function, or subroutine

When an IF statement is executed, the logical expression is evaluated first. If the value is true, the statement is executed. If the value is false, the statement is not executed and control transfers to the next statement in the program.

Example

The following examples show valid IF statements:

```
IF (J.GT.4 .OR. J.LT.1) GO TO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K) * (-1.5D0)

IF (ENDRUN) CALL EXIT
```

The following shows another example:

```
   USE IFPORT
   INTEGER(4) istat, errget
   character(inchar)
   real x
   istat = getc(inchar)
   IF (istat) errget = -1
   ...
!   IF (x .GT. 2.3) call new_subr(x)
   ...
```

See Also

- H to I
- IF Construct
- Execution Control

IF Construct

Statement: *Conditionally executes one block of constructs or statements depending on the evaluation of a logical expression. (This construct was called a block IF statement in FORTRAN 77.)*

Syntax

name:] IF (*expr*) THEN

block

[ELSE IF (*expr*) THEN [*name*]

block]

[ELSE [*name*]

block]

END IF [*name*]

name	(Optional) Is the name of the IF construct.
expr	Is a scalar logical expression enclosed in parentheses.

block Is a sequence of zero or more statements or constructs.

Description

If a construct name is specified at the beginning of an IF THEN statement, the same name must appear in the corresponding END IF statement. The same construct name must not be used for different named constructs in the same scoping unit.

Depending on the evaluation of the logical expression, one block or no block is executed. The logical expressions are evaluated in the order in which they appear, until a true value is found or an ELSE or END IF statement is encountered.

Once a true value is found or an ELSE statement is encountered, the block immediately following it is executed and the construct execution terminates.

If none of the logical expressions evaluate to true and no ELSE statement appears in the construct, no block in the construct is executed and the construct execution terminates.

NOTE. No additional statement can be placed after the IF THEN statement in a block IF construct. For example, the following statement is invalid in the block IF construct:

```
IF (e) THEN I = J
```

This statement is translated as the following logical IF statement:

```
IF (e) THENI = J
```

You cannot use branching statements to transfer control to an ELSE IF statement or ELSE statement. However, you can branch to an END IF statement from within the IF construct.

The following figure shows the flow of control in IF constructs:

2878

Figure 62: Flow of Control in IF Constructs

You can include an IF construct in the statement block of another IF construct, if the nested IF construct is completely contained within a statement block. It cannot overlap statement blocks.

Example

The following example shows the simplest form of an IF construct:

```
Form                  Example

IF (expr) THEN        IF (ABS(ADJU) .GE. 1.0E-6) THEN

  block                 TOTERR = TOTERR + ABS(ADJU)

                        QUEST = ADJU/FNDVAL

END IF                END IF
```

This construct conditionally executes the block of statements between the IF THEN and the END IF statements.

The following shows another example:

```
!  Simple block IF:

   IF (i .LT. 10) THEN

      ! the next two statements are only executed if i < 10

      j = i

      slice = TAN (angle)

   END IF
```

The following example shows a named IF construct:

```
  BLOCK_A: IF (D > 0.0) THEN        ! Initial statement for named construct

    RADIANS = ACOS(D)               ! These two statements

    DEGREES = ACOSD(D)              !       form a block

  END IF BLOCK_A                    ! Terminal statement for named construct
```

The following example shows an IF construct containing an ELSE statement:

```
Form                  Example

IF (expr) THEN        IF (NAME .LT. 'N') THEN
  block1                IFRONT = IFRONT + 1
                        FRLET(IFRONT) = NAME(1:2)
ELSE                  ELSE
  block2                IBACK = IBACK + 1
END IF                END IF
```

Block1 consists of all the statements between the IF THEN and ELSE statements. Block2 consists of all the statements between the ELSE and the END IF statements.

If the value of the character variable NAME is less than `'N '`, block1 is executed. If the value of NAME is greater than or equal to `'N '`, block2 is executed.

The following example shows an IF construct containing an ELSE IF THEN statement:

```
Form                      Example  IF (expr) THEN           IF (A .GT. B) THEN
  block1                    D = B
                            F = A - B
ELSE IF (expr) THEN       ELSE IF (A .GT. B/2.) THEN
  block2                    D = B/2.
                            F = A - B/2.
END IF                    END IF
```

If A is greater than B, block1 is executed. If A is not greater than B, but A is greater than B/2, block2 is executed. If A is not greater than B and A is not greater than B/2, neither block1 nor block2 is executed. Control transfers directly to the next executable statement after the END IF statement.

The following shows another example:

```
!  Block IF with ELSE IF statements:
   IF (j .GT. 1000) THEN
      ! Statements here are executed only if J > 1000
   ELSE IF (j .GT. 100) THEN
      ! Statements here are executed only if J > 100 and j <= 1000
   ELSE IF (j .GT. 10) THEN
      ! Statements here are executed only if J > 10 and j <= 100
   ELSE
      ! Statements here are executed only if j <= 10
   END IF
```

The following example shows an IF construct containing several ELSE IF THEN statements and an ELSE statement:

Form	**Example**
IF (expr) THEN	IF (A .GT. B) THEN
block1	D = B
	F = A - B
ELSE IF (expr) THEN	ELSE IF (A .GT. C) THEN
block2	D = C
	F = A - C
ELSE IF (expr) THEN	ELSE IF (A .GT. Z) THEN
block3	D = Z
	F = A - Z
ELSE	ELSE
block4	D = 0.0
	F = A
END IF	END IF

If A is greater than B, block1 is executed. If A is not greater than B but is greater than C, block2 is executed. If A is not greater than B or C but is greater than Z, block3 is executed. If A is not greater than B, C, or Z, block4 is executed.

The following example shows a nested IF construct:

Form	**Example**
IF (expr) THEN	IF (A .LT. 100) THEN
block1	INRAN = INRAN + 1
IF (expr2) THEN	IF (ABS(A-AVG) .LE. 5.) THEN
block1a	INAVG = INAVG + 1
ELSE	ELSE
block1b	OUTAVG = OUTAVG + 1
END IF	END IF
ELSE	ELSE
block2	OUTRAN = OUTRAN + 1
END IF	END IF

If A is less than 100, the code immediately following the IF is executed. This code contains a nested IF construct. If the absolute value of A minus AVG is less than or equal to 5, block1a is executed. If the absolute value of A minus AVG is greater than 5, block1b is executed.

If A is greater than or equal to 100, block2 is executed, and the nested IF construct (in block1) is not executed.

The following shows another example:

```
!  Nesting of constructs and use of an ELSE statement following
!  a block IF without intervening ELSE IF statements:
   IF (i .LT. 100) THEN
      ! Statements here executed only if i < 100
      IF (j .LT. 10) THEN
      ! Statements here executed only if i < 100 and j < 10
      END IF
      ! Statements here executed only if i < 100
   ELSE
      ! Statements here executed only if i >= 100
      IF (j .LT. 10) THEN
      ! Statements here executed only if i >= 100 and j < 10
      END IF
      ! Statements here executed only if i >= 100
   END IF
```

See Also

- H to I
- Execution Control
- IF - Logical
- IF - Arithmetic

IF Directive Construct

General Compiler Directive: *A conditional compilation construct that begins with an IF or IF DEFINED directive. IF tests whether a logical expression is .TRUE. or .FALSE.. IF DEFINED tests whether a symbol has been defined.*

Syntax

```
cDEC$ IF (expr) -or- cDEC$ IF DEFINED (name)
```

```
   block
[cDEC$ ELSEIF (expr)
   block] ...
[cDEC$ ELSE
   block]
cDEC$ ENDIF
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
expr	Is a logical expression that evaluates to .TRUE. or .FALSE..
name	Is the name of a symbol to be tested for definition.
block	Are executable statements that are compiled (or not) depending on the value of logical expressions in the IF directive construct.

The IF and IF DEFINED directive constructs end with an ENDIF directive and can contain one or more ELSEIF directives and at most one ELSE directive. If the logical condition within a directive evaluates to .TRUE. at compilation, and all preceding conditions in the IF construct evaluate to .FALSE., then the statements contained in the directive block are compiled.

A *name* can be defined with a DEFINE directive, and can optionally be assigned an integer value. If the symbol has been defined, with or without being assigned a value, IF DEFINED (name) evaluates to .TRUE.; otherwise, it evaluates to .FALSE..

If the logical condition in the IF or IF DEFINED directive is .TRUE., statements within the IF or IF DEFINED block are compiled. If the condition is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If the logical expression in an ELSEIF directive is .TRUE., statements within the ELSEIF block are compiled. If the expression is .FALSE., control transfers to the next ELSEIF or ELSE directive, if any.

If control reaches an ELSE directive because all previous logical conditions in the IF construct evaluated to .FALSE., the statements in an ELSE block are compiled unconditionally.

You can use any Fortran logical or relational operator or symbol in the logical expression of the directive, including: .LT., <, .GT., >, .EQ., ==, .LE., <=, .GE., >=, .NE., /=, .EQV., .NEQV., .NOT., .AND., .OR., and .XOR.. The logical expression can be as complex as you like, but the whole directive must fit on one line.

Example

```
!  When the following code is compiled and run,
!  the output is:
!  Or this compiled if all preceding conditions .FALSE.
!
!DEC$ DEFINE flag=3
!DEC$ IF (flag .LT. 2)
   WRITE (*,*) "This is compiled if flag less than 2."
!DEC$ ELSEIF (flag >= 8)
   WRITE (*,*) "Or this compiled if flag greater than &
               or equal to 8."
!DEC$ ELSE
   WRITE (*,*) "Or this compiled if all preceding &
               conditions .FALSE."
!DEC$ ENDIF
END
```

See Also

- H to I
- DEFINE and UNDEFINE
- IF Construct
- General Compiler Directives

IF DEFINED Directive

See IF Directive Construct.

IFIX

Elemental Intrinsic Function (Generic): *Converts a single-precision real argument to an integer by truncating. See INT.*

IFLOATI, IFLOATJ

Portability Functions: *Convert an integer to single-precision real type.*

Module

USE IFPORT

Syntax

```
result = IFLOATI (i)
result = IFLOATJ (j)
```

i	(Input) Must be of type INTEGER(2).
j	(Input) Must be of type INTEGER(4).

Results

The result type is single-precision real (REAL(4) or REAL*4).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- DFLOAT

ILEN

Inquiry Intrinsic Function (Generic): *Returns the length (in bits) of the two's complement representation of an integer.*

Syntax

```
result = ILEN (i)
```

i	Must be of type integer.

Results

The result type is the same as *i*. The result value is (LOG_2(*i* + 1)) if *i* is not negative; otherwise, the result value is (LOG_2(- *i*)).

Example

```
ILEN (4) has the value 3.

ILEN (-4) has the value 2.
```

IMAGESIZE, IMAGESIZE_W (W*32, W*64)

Graphics Functions: *Return the number of bytes needed to store the image inside the specified bounding rectangle. IMAGESIZE is useful for determining how much memory is needed for a call to GETIMAGE.*

Module

USE IFQWIN

Syntax

```
result = IMAGESIZE (x1,y1,x2,y2)

result = IMAGESIZE_W (wx1,wy1,wx2,wy2)
```

x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of image.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of image.

wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of image.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of image.

Results

The result type is INTEGER(4). The result is the storage size of an image in bytes.

IMAGESIZE defines the bounding rectangle in viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). IMAGESIZE_W defines the bounding rectangle in window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

IMAGESIZE_W defines the bounding rectangle in terms of window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

See the example in GETIMAGE.

See Also

- H to I
- GETIMAGE
- GRSTATUS
- PUTIMAGE

Building Applications: Transferring Images in Memory

IMPLICIT

Statement: *Overrides the default implicit typing rules for names. (The default data type is INTEGER for names beginning with the letters I through N, and REAL for names beginning with any other letter.)*

Syntax

The IMPLICIT statement takes one of the following forms:

```
IMPLICIT type(a[,a]...)[,type(a[,a]...)]...
```

```
IMPLICIT NONE
```

type	Is a data type specifier (CHARACTER*(*) is not allowed).
a	Is a single letter, a dollar sign ($),or a range of letters in alphabetical order. The form for a range of letters is a_1-a_2, where the second letter follows the first alphabetically (for example, A-C).

The dollar sign can be used at the end of a range of letters, since IMPLICIT interprets the dollar sign to alphabetically follow the letter Z. For example, a range of X-$ would apply to identifiers beginning with the letters X, Y, Z, or $.

In Intel® Fortran, the parentheses around the list of letters are optional.

Description

The IMPLICIT statement assigns the specified data type (and kind parameter) to all names that have no explicit data type and begin with the specified letter or range of letters. It has no effect on the default types of intrinsic procedures.

When the data type is CHARACTER*len, *len* is the length for character type. The *len* is an unsigned integer constant or an integer specification expression enclosed in parentheses. The range for *len* is 1 to 2**31-1 on IA-32 architecture; 1 to 2**63-1 on Intel® 64 architecture and IA-64 architecture.

Names beginning with a dollar sign ($) are implicitly INTEGER.

The IMPLICIT NONE statement disables all implicit typing defaults. When IMPLICIT NONE is used, all names in a program unit must be explicitly declared. An IMPLICIT NONE statement must precede any PARAMETER statements, and there must be no other IMPLICIT statements in the scoping unit.

NOTE. To receive diagnostic messages when variables are used but not declared, you can specify compiler option warn declarations instead of using IMPLICIT NONE.

The following IMPLICIT statement represents the default typing as specified by the Fortran Standard for names when they are not explicitly typed:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

Example

The following are examples of the IMPLICIT statement:

```
IMPLICIT DOUBLE PRECISION (D) IMPLICIT COMPLEX (S,Y), LOGICAL(1) (L,A-C)
```

```
IMPLICIT CHARACTER*32 (T-V)

IMPLICIT CHARACTER*2 (W)

IMPLICIT TYPE(COLORS) (E-F), INTEGER (G-H)
```

The following shows another example:

```
SUBROUTINE FF (J)

IMPLICIT INTEGER (a-b), CHARACTER*(J+1) (n), TYPE(fried) (c-d)

TYPE fried

INTEGER e, f

REAL g, h

END TYPE

age = 10 ! integer

name = 'Paul' ! character

c%e = 1 ! type fried, integer component
```

See Also

- H to I
- Data Types, Constants, and Variables
- warn declarations compiler option

IMPORT

Statement: *Makes host entities accessible in the interface body of an interface block. It takes the following form:*

Syntax

```
IMPORT [[::] import-name-list]
```

import-name-list (Input) Is the name of one or more entities in the host scoping unit.

An IMPORT statement can appear only in an interface body. Each of the named entities must be explicitly declared before the interface body containing the IMPORT statement.

If *import-name-list* is not specified, all of the accessible named entities in the host scoping unit are imported.

The default is that all accessible symbols are imported. This statement can optionally name only those symbols you choose.

The names of imported entities must not appear in any context that causes the host entity to be inaccessible.

Example

The following example shows how the IMPORT statement can be applied.

```
module mymod
type mytype
  integer comp
end type mytype
interface
  subroutine sub (arg)
    import
    type(mytype) :: arg
  end subroutine sub
end interface
end module mymod
```

INCHARQQ (W*32, W*64)

QuickWin Function: *Reads a single character input from the keyboard and returns the ASCII value of that character without any buffering.*

Module

USE IFQWIN

Syntax

```
result = INCHARQQ( )
```

Results

The result type is INTEGER(2). The result is the ASCII key code.

The keystroke is read from the child window that currently has the focus. You must call INCHARQQ before the keystroke is made (INCHARQQ does not read the keyboard buffer). This function does not echo its input. For function keys, INCHARQQ returns 0xE0 as the upper 8 bits, and the ASCII code as the lower 8 bits.

For direction keys, INCHARQQ returns 0xF0 as the upper 8 bits, and the ASCII code as the lower 8 bits. To allow direction keys to be read, you must use the PASSDIRKEYSQQ function. The escape characters (the upper 8 bits) are different from those of GETCHARQQ. Note that console applications do not need, and cannot use PASSDIRKEYSQQ.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
use IFQWIN
integer*4 res
integer*2 exchar
character*1 ch, ch1
Print *,"Type X to exit, S to scroll, D to pass Direction keys"
123 continue
exchar = incharqq()
! check for escapes
! 0xE0 0x?? is a function key
! 0xF0 0x?? is a direction key
ch = char(rshift(exchar,8) .and. Z'00FF')
ch1= char(exchar .and. Z'00FF')
if (ichar(ch) .eq. 224) then
  print *,"function key  = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
endif
if (ichar(ch) .eq. 240) then
  print *,"direction key = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
endif
print *,"other key  = ",ichar(ch),"  ",ichar(ch1),"  ",ch1
if(ch1 .eq. 'S') then
  res = passdirkeysqq(.false.)
  print *, "Entering Scroll mode"
endif
if(ch1 .eq. 'D')  then
  res = passdirkeysqq(.true.)
```

```
  print *, "Entering Direction keys mode"
endif
if(ch1 .ne. 'X')  go to 123
end
```

See Also

- H to I
- GETCHARQQ
- READ
- MBINCHARQQ
- GETC
- PASSDIRKEYSQQ

Building Applications: Using QuickWin Overview

Building Applications: Blocking Procedures

INCLUDE

Statement: *Directs the compiler to stop reading statements from the current file and read statements in an included file or text module.*

Syntax

The INCLUDE line takes the following form:

```
INCLUDE 'filename[/[NO]LIST]'
```

filename	Is a character string specifying the name of the file to be included; it must not be a named constant. The form of the file name must be acceptable to the operating system, as described in your system documentation.
[NO]LIST	Specifies whether the incorporated code is to appear in the compilation source listing. In the listing, a number precedes each incorporated statement. The number indicates the "include" nesting depth of the code. The default is /NOLIST. /LIST and /NOLIST must be spelled completely. You can only use /[NO]LIST if you specify compiler option vms (which sets OpenVMS defaults).

Description

An INCLUDE line can appear anywhere within a scoping unit. The line can span more than one source line, but no other statement can appear on the same line. The source line cannot be labeled.

An included file or text module cannot begin with a continuation line, and each Fortran statement must be completely contained within a single file.

An included file or text module can contain any source text, but it cannot begin or end with an incomplete Fortran statement.

The included statements, when combined with the other statements in the compilation, must satisfy the statement-ordering restrictions shown in Statements.

Included files or text modules can contain additional INCLUDE lines, but they must not be recursive. INCLUDE lines can be nested until system resources are exhausted.

When the included file or text module completes execution, compilation resumes with the statement following the INCLUDE line.

You can use modules instead of include files to achieve encapsulation of related data types and procedures. For example, one module can contain derived type definitions as well as special operators and procedures that apply to those types. For information on how to use modules, see Program Units and Procedures.

Example

In the following example, a file named COMMON.FOR (in the current working directory) is included and read as input.

Figure 63: Including Text from a File

Main Program File

```
PROGRAM
  INCLUDE 'COMMON.FOR'
  REAL, DIMENSION(M) :: Z
  CALL CUBE
  DO I = 1, M
    Z(I) = X(I) + SQRT(Y(I))
    ...
  END DO
END
SUBROUTINE CUBE
  INCLUDE 'COMMON.FOR'
  DO I=1,M
    X(I) = Y(I)**3
  END DO
  RETURN
END
```

COMMON.FOR File

```
INTEGER, PARAMETER :: M=100
REAL, DIMENSION(M) :: X, Y
COMMON X, Y
```

The file COMMON.FOR defines a named constant M, and defines arrays X and Y as part of blank common.

The following example program declares its common data in an include file. The contents of the file INCLUDE.INC are inserted in the source code in place of every INCLUDE 'INCLUDE.INC' line. This guarantees that all references to common storage variables are consistent.

```
INTEGER i
REAL x
```

```
INCLUDE 'INCLUDE.INC'
DO i = 1, 5
  READ (*, '(F10.5)') x
  CALL Push (x)
END DO
```

See Also

- H to I
- MODULE
- USE

INDEX

Elemental Intrinsic Function (Generic):
Returns the starting position of a substring within a string.

Syntax

`result = INDEX (string, substring [,back] [, kind])`

string	(Input) Must be of type character.
substring	(Input) Must be of type character.
back	(Input; optional) Must be of type logical.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* does not appear (or appears with the value false), the value returned is the minimum value of I such that *string*(I : I + LEN (*substring*) - 1) = *substring*(or zero if there is no such value). If LEN (*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, 1 is returned.

If *back* appears with the value true, the value returned is the maximum value of I such that *string*(I : I + LEN (*substring*) - 1) = *substring* (or zero if there is no such value). If LEN(*string*) < LEN (*substring*), zero is returned. If LEN (*substring*) = zero, LEN (*string*) + 1 is returned.

Specific Name	Argument Type	Result Type
	CHARACTER	INTEGER(1)
	CHARACTER	INTEGER(2)
INDEX [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

[1]The setting of compiler options specifying integer size can affect this function.

Example

INDEX ('FORTRAN', 'O', BACK = .TRUE.) has the value 2.

INDEX ('XXXX', " ", BACK = .TRUE.) has the value 5.

The following shows another example:

```
I = INDEX('banana','an', BACK = .TRUE.) ! returns 4

I = INDEX('banana', 'an') ! returns 2
```

See Also

- H to I
- SCAN

INITIALIZEFONTS (W*32, W*64)

Graphics Function: *Initializes Windows* fonts.*

Module

USE IFQWIN

Syntax

```
result = INITIALIZEFONTS( )
```

Results

The result type is INTEGER(2). The result is the number of fonts initialized.

All fonts on Windows systems become available after a call to INITIALIZEFONTS. Fonts must be initialized with INITIALIZEFONTS before any other font-related library function (such as GETFONTINFO, GETGTEXTEXTENT, SETFONT, OUTGTEXT) can be used. For more information, see *Building Applications*: Using Fonts from the Graphics Library.

The font functions affect the output of OUTGTEXT only. They do not affect other Fortran I/O functions (such as WRITE) or graphics output functions (such as OUTTEXT).

For each window you open, you must call INITIALIZEFONTS before calling SETFONT. INITIALIZEFONTS needs to be executed after each new child window is opened in order for a subsequent SETFONT call to be successful.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! build as a QuickWin or Standard Graphics App.

USE IFQWIN

INTEGER(2) numfonts

numfonts = INITIALIZEFONTS()

WRITE (*,*) numfonts

END
```

See Also

- H to I
- SETFONT
- OUTGTEXT

Building Applications: Initializing Fonts

INITIALSETTINGS (W*32, W*64)

QuickWin Function: *Initializes QuickWin.*

Module

USE IFQWIN

Syntax

```
result = INITIALSETTINGS( )
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can change the initial appearance of an application's default frame window and menus by defining an INITIALSETTINGS function. Do not use INITIALSETTINGS to open or size child windows.

If no user-defined INITIALSETTINGS function is supplied, QuickWin calls a predefined INITIALSETTINGS routine to control the default frame window and menu appearance. You do not need to call INITIALSETTINGS if you define it, since it will be called automatically during initialization.

For more information, see *Building Applications*: Controlling the Initial Menu and Frame Window.

Compatibility

QUICKWIN GRAPHICS WINDOWS LIB

See Also

- H to I
- APPENDMENUQQ
- INSERTMENUQQ
- DELETEMENUQQ
- SETWSIZEQQ

Building Applications: Using QuickWin Overview

Building Applications: Program Control of Menus

INMAX

Portability Function: *Returns the maximum positive value for an integer.*

Module

USE IFPORT

Syntax

```
result = INMAX (i)
```

i (Input) INTEGER(4).

Results

The result type is INTEGER(4). The result is the maximum 4-byte signed integer value for the argument.

INQFOCUSQQ (W*32, W*64)

QuickWin Function: *Determines which window has the focus.*

Module

USE IFQWIN

Syntax

```
result = INQFOCUSQQ (unit)
```

unit (Output) INTEGER(4). Unit number of the window that has the I/O focus.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero. The function fails if the window with the focus is associated with a closed unit.

Unit numbers 0, 5, and 6 refer to the default window only if the program has not specifically opened them. If these units have been opened and connected to windows, they are automatically reconnected to the console once they are closed.

The window with focus is always in the foreground. Note that the window with the focus is not necessarily the active window (the one that receives graphical output). A window can be made active without getting the focus by calling SETACTIVEQQ.

A window has focus when it is given the focus by FOCUSQQ, when it is selected by a mouse click, or when an I/O operation other than a graphics operation is performed on it, unless the window was opened with IOFOCUS=.FALSE.. The IOFOCUS specifier determines whether a window receives focus when an I/O statement is executed on that unit. For example:

```
OPEN (UNIT = 10, FILE = 'USER', IOFOCUS = .TRUE.)
```

By default IOFOCUS=.TRUE., except for child windows opened with as unit *. If IOFOCUS=.TRUE., the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- H to I
- FOCUSQQ

Building Applications: Using QuickWin Overview

INQUIRE

Statement: *Returns information on the status of specified properties of a file, logical unit, or directory. It takes one of the following forms:*

Syntax

Inquiring by File:

```
INQUIRE (FILE=name[, ERR=label] [, IOSTAT=i-var] [, DEFAULTFILE=def] slist)
```

Inquiring by Unit:

```
INQUIRE ([UNIT=]io-unit [, ERR=label] [, IOSTAT=i-var] slist)
```

Inquiring by Directory:

```
INQUIRE (DIRECTORY=dir, EXIST=ex [, DIRSPEC=dirspec] [, ERR=label] [,
IOSTAT=i-var])
```

Inquiring by Output List:

```
INQUIRE (IOLENGTH=len) out-item-list
```

name	Is a scalar default character expression specifying the name of the file for inquiry. For more information, see FILE Specifier and STATUS Specifier.
label	Is the label of the branch target statement that receives control if an error occurs. For more information, see Branch Specifiers.

i-var Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs. For more information, see I/O Status Specifier.

def Is a scalar default character expression specifying a default file pathname string. (For more information, see the DEFAULTFILE specifier.)

slist Is one or more of the following inquiry specifiers (each specifier can appear only once):

ACCESS	DELIM	NAMED	READ
ACTION	DIRECT	NEXTREC	READWRITE
ASYNCHRONOUS	EXIST	NUMBER	RECL
BINARY	FORM	OPENED	RECORDTYPE
BLANK	FORMATTED	ORGANIZATION	SEQUENTIAL
BLOCKSIZE	ID	PAD	SHARE
BUFFERED	IOFOCUS	PENDING	UNFORMATTED
CARRIAGECONTROL	MODE	POS	WRITE
CONVERT	NAME	POSITION	

io-unit Is an external unit specifier.
The unit does not have to exist, nor does it need to be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

dir Is a scalar default character expression specifying the name of the directory for inquiry. If you are inquiring by directory, it must be present.

ex Is a scalar default logical variable that is assigned the value .TRUE. if *dir* names a directory that exists; otherwise, *ex* is assigned the value .FALSE.. If you are inquiring by directory, it must be present. For more information, see the EXIST Specifier.

dirspec	Is a scalar default character variable that is assigned the value of the full directory specification of *dir* if *ex* is assigned the value .TRUE.. This specifier can only be used when inquiring by directory.
len	(Output) Is a scalar integer variable that is assigned a value corresponding to the length of an unformatted, direct-access record resulting from the use of the *out-item-list* in a WRITE statement. The value is suitable to use as a RECL specifier value in an OPEN statement that connects a file for unformatted, direct access. The unit of the value is 4-byte longwords, by default. However, if you specify compiler option assume byterecl, the unit is bytes.
out-item-list	(Output) Is a list of one or more output items (see I/O Lists).

Description

The control specifiers ([UNIT=] *io-unit*, ERR= *label*, and IOSTAT= *i-var*) and inquiry specifiers can appear anywhere within the parentheses following INQUIRE. However, if the UNIT keyword is omitted, the *io-unit* must appear first in the list.

An INQUIRE statement can be executed before, during, or after a file is connected to a unit. The specifier values returned are those that are current when the INQUIRE statement executes.

To get file characteristics, specify the INQUIRE statement after opening the file.

Example

The following are examples of INQUIRE statements:

```
INQUIRE (FILE='FILE_B', EXIST=EXT)

INQUIRE (4, FORM=FM, IOSTAT=IOS, ERR=20)

INQUIRE (IOLENGTH=LEN) A, B
```

In the last statement, you can use the length returned in LEN as the value for the RECL specifier in an OPEN statement that connects a file for unformatted direct access. If you have already specified a value for RECL, you can check LEN to verify that A and B are less than or equal to the record length you specified.

The following shows another example:

```
!    This program prompts for the name of a data file.
!    The INQUIRE statement then determines whether
!    the file exists. If it does not, the program
!    prompts for another file name.
      CHARACTER*12 fname
      LOGICAL exists
!     Get the name of a file:
100   WRITE (*, '(1X, A\)') 'Enter the file name: '
      READ (*, '(A)') fname
!     INQUIRE about file's existence:
      INQUIRE (FILE = fname, EXIST = exists)
      IF (.NOT. exists) THEN
        WRITE (*,'(2A/)') ' >> Cannot find file ', fname
        GOTO 100
      END IF
      END
```

See Also

- H to I
- OPEN statement
- UNIT control specifier
- ERR control specifier
- IOSTAT control specifier
- RECL specifier in OPEN statements
- FILE specifier in OPEN statements
- DEFAULTFILE specifier in OPEN statements
- assume minus0 compiler option

INSERTMENUQQ (W*32, W*64)

QuickWin Function: *Inserts a menu item into a QuickWin menu and registers its callback routine.*

Module

USE IFQWIN

Syntax

`result = INSERTMENUQQ (menuID,itemID,flag,text,routine)`

menuID (Input) INTEGER(4). Identifies the menu in which the item is inserted, starting with 1 as the leftmost menu.

itemID (Input) INTEGER(4). Identifies the position in the menu where the item is inserted, starting with 0 as the top menu item.

flag (Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see Results section below). The following constants are available:

- $MENUGRAYED - Disables and grays out the menu item.
- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

text (Input) Character*(*). Menu item name. Must be a null-terminated C string, for example, words of text'C.

routine (Input) EXERNAL. Callback subroutine that is called if the menu item is selected. All routines ake a single LOGICAL parameter that indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.

- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.
- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.
- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.
- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE.

Menus and menu items must be defined in order from left to right and top to bottom. For example, INSERTMENUQQ fails if you try to insert menu item 7 when 5 and 6 are not defined yet. For a top-level menu item, the callback routine is ignored if there are subitems under it.

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

You can create quick-access keys in the text strings you pass to INSERTMENUQQ as *text* by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, *text* should be "P&rint". Quick-access keys allow users of your program to activate that menu item with the key combination ALT+QUICK-ACCESS-KEY(ALT+Rin the example) as an alternative to selecting the item with the mouse.

For more information on customizing QuickWin menus, see *Building Applications*: Using QuickWin Overview.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
! build as a QuickWin App.
USE IFQWIN
LOGICAL(4) status
! insert new item into Menu 5 (Window)
status= INSERTMENUQQ(5, 5, $MENUCHECKED, 'New Item'C, &
                     WINSTATUS)
! insert new menu in position 2
status= INSERTMENUQQ(2, 0, $MENUENABLED, 'New Menu'C, &
                     WINSAVE)
END
```

See Also

- H to I
- APPENDMENUQQ
- DELETEMENUQQ
- MODIFYMENUFLAGSQQ
- MODIFYMENUROUTINEQQ
- MODIFYMENUSTRINGQQ

INT

Elemental Intrinsic Function (Generic):
Converts a value to integer type.

Syntax

```
result = INT (a[,kind])
```

a	(Input) Must be of type integer, real, or complex.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

The result value depends on the type and absolute value of *a*, as follows:

- If *a* is of type integer, INT(*a*) = *a*.
- If *a* is of type real and | *a* | < 1, INT(*a*) has the value zero.

  If *a* is of type real and | *a* | >=1, INT(*a*) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *a* and whose sign is the same as the sign of *a*.
- If *a* is of type complex, INT(*a*) = *a* is the value obtained by applying the preceding rules (for a real argument) to the real part of *a*.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)
	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8)	INTEGER(8)
IJINT	INTEGER(4)	INTEGER(2)

Specific Name [1]	Argument Type	Result Type
IIFIX[2]	REAL(4)	INTEGER(2)
IINT	REAL(4)	INTEGER(2)
IFIX [3, 4]	REAL(4)	INTEGER(4)
JFIX	INTEGER(1), INTEGER(2), INTEGER(4),	INTEGER(4)
	INTEGER(8), REAL(4), REAL(8), REAL(16),	
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	
INT [5, 6, 7]	REAL(4)	INTEGER(4)
KIFIX	REAL(4)	INTEGER(8)
KINT	REAL(4)	INTEGER(8)
IIDINT	REAL(8)	INTEGER(2)
IDINT [6, 8]	REAL(8)	INTEGER(4)
KIDINT	REAL(8)	INTEGER(8)
IIQINT	REAL(16)	INTEGER(2)
IQINT[6, 9]	REAL(16)	INTEGER(4)
KIQINT	REAL(16)	INTEGER(8)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)

Specific Name [1]	Argument Type	Result Type
	COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)
INT1[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(1)
INT2[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(2)
INT4[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(4)
INT8[10]	INTEGER(1), INTEGER(2), INTEGER(4), INTEGER(8), REAL(4), REAL(8), REAL(16), COMPLEX(4), COMPLEX(8), COMPLEX(16)	INTEGER(8)

[1]These specific functions cannot be passed as actual arguments.

[2]This function can also be specified as HFIX.

Specific Name [1]	Argument Type	Result Type

[3]The setting of compiler options specifying integer size or real size can affect IFIX.

[4]For compatibility with older versions of Fortran, IFIX is treated as a generic function.

[5]Or JINT.

[6]The setting of compiler options specifying integer size can affect INT, IDINT, and IQINT.

[7]Or JIFIX.

[8]Or JIDINT. For compatibility with older versions of Fortran, IDINT can also be specified as a generic function.

[9]Or JIQINT. For compatibility with older versions of Fortran, IQINT can also be specified as a generic function.

[10]For compatibility, these functions can also be specified as generic functions.

Example

INT (-4.2) has the value -4.

INT (7.8) has the value 7.

See Also

- H to I
- NINT
- AINT
- ANINT
- REAL
- DBLE
- SNGL

INTC

Portability Function: *Converts an INTEGER(4) argument to INTEGER(2) type.*

Module

USE IFPORT

Syntax

```
result = INTC (i)
```

i (Input) INTEGER(4). A value or expression.

Results

The result type is INTEGER(2). The result is the value of *i* with type INTEGER(2). Overflow is ignored.

INT_PTR_KIND

Inquiry Intrinsic Function (Specific): *Returns the INTEGER KIND that will hold an address. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax

```
result = INT_PTR_KIND( )
```

Results

The result type is default integer. The result is a scalar with the value equal to the value of the kind parameter of the integer data type that can represent an address on the host platform.

The result value is 4 on IA-32 architecture; 8 on Intel® 64 architecture and IA-64 architecture.

Example

```
REAL A(100)
POINTER (P, A)
INTEGER (KIND=INT_PTR_KIND()) SAVE_P
P = MALLOC (400)
SAVE_P = P
```

INTEGER Statement

Statement: *Specifies the INTEGER data type.*

Syntax

```
INTEGER

INTEGER([KIND=] n)

INTEGER*n
```

n Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the integer has the kind specified. If a kind parameter is not specified, integer constants are interpreted as follows:

- If the integer constant is within the default integer kind range, the kind is default integer.
- If the integer constant is outside the default integer kind range, the kind of the integer constant is the smallest integer kind which holds the constant.

The default kind can also be changed by using the INTEGER directive or compiler options specifying integer size.

Example

```
! Entity-oriented declarations:

INTEGER, DIMENSION(:), POINTER :: days, hours

INTEGER (2) :: k=4

INTEGER (2), PARAMETER :: limit=12

! Attribute-oriented declarations:

INTEGER days, hours

INTEGER (2):: k=4, limit

DIMENSION days(:), hours(:)

POINTER days, hours

PARAMETER (limit=12)
```

See Also

- H to I

- INTEGER Directive
- Integer Data Types
- Integer Constants

INTEGER Directive

General Compiler Directive: *Specifies the default integer kind.*

Syntax

```
cDEC$ INTEGER:{ 2 | 4 | 8 }
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The INTEGER directive specifies a size of 2 (KIND=2), 4 (KIND=4), or 8 (KIND=8) bytes for default integer numbers.

When the INTEGER directive is in effect, all default integer variables are of the kind specified. Only numbers specified or implied as INTEGER without KIND are affected.

The INTEGER directive can only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module or a block data program unit. INTEGER cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

The default logical kind is the same as the default integer kind. So, when you change the default integer kind you also change the default logical kind.

Example

```
INTEGER i            ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )
WRITE(*,*) KIND(i)   ! still a 4-byte integer
                     !   not affected by setting in subroutine
END
SUBROUTINE INTEGER2( )
  !DEC$ INTEGER:2
  INTEGER j          ! a 2-byte integer
  WRITE(*,*) KIND(j)
END SUBROUTINE
```

See Also

- H to I
- INTEGER
- REAL Directive
- General Compiler Directives
- Integer Data Types
- Integer Constants

INTEGERTORGB (W*32, W*64)

QuickWin Subroutine: *Converts an RGB color value into its red, green, and blue components.*

Module

USE IFQWIN

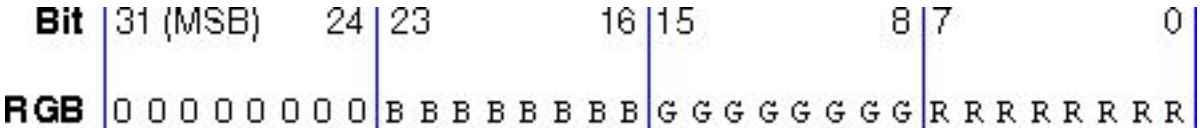
Syntax

CALL INTEGERTORGB (*rgb*,*red*,*green*,*blue*)

rgb	(Input) INTEGER(4). RGB color value whose red, green, and blue components are to be returned.

red	(Output) INTEGER(4). Intensity of the red component of the RGB color value.
green	(Output) INTEGER(4). Intensity of the green component of the RGB color value.
blue	(Output) INTEGER(4). Intensity of the blue component of the RGB color value.

INTEGERTORGB separates the four-byte RGB color value into the three components as follows:



Compatibility

QUICKWIN GRAPHICS WINDOWS LIB

Example

```
! build as a QuickWin App.

USE IFQWIN

INTEGER(4) r, g, b

CALL INTEGERTORGB(2456, r, g, b)

write(*,*) r, g, b

END
```

See Also

- H to I
- RGBTOINTEGER
- GETCOLORRGB
- GETBKCOLORRGB
- GETPIXELRRGB
- GETPIXELSRGB
- GETTEXTCOLORRGB

Building Applications: Using QuickWin Overview

INTENT

Statement and Attribute: *Specifies the intended use of one or more dummy arguments.*

Syntax

The INTENT attribute can be specified in a type declaration statement or an INTENT statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,]  INTENT (intent-spec) [, att-ls] :: d-arg[, d-arg]...
```

Statement:

```
INTENT (intent-spec) [::] d-arg[, d-arg] ...
```

type — Is a data type specifier.

att-ls — Is an optional list of attribute specifiers.

intent-spec — Is one of the following specifiers:

- *IN* — Specifies that the dummy argument will be used only to provide data to the procedure. The dummy argument must not be redefined (or become undefined) during execution of the procedure.
- *OUT* — Specifies that the dummy argument will be used to pass data from the procedure back to the calling program. The dummy argument is undefined on entry and must be defined before it is referenced in the procedure.
  Any associated actual argument must be definable.
- *INOUT* — Specifies that the dummy argument can both provide data to the procedure and return data to the calling program.
  Any associated actual argument must be definable.

d-arg — Is the name of a dummy argument or dummy pointer. It cannot be a dummy procedure.

Description

The INTENT statement can only appear in the specification part of a subprogram or interface body.

If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

If a function specifies a defined operator, the dummy arguments must have intent IN.

If a subroutine specifies defined assignment, the first argument must have intent OUT or INOUT, and the second argument must have intent IN.

A non-pointer dummy argument with intent IN (or a subobject of such a dummy argument) must *not* appear as any of the following:

- A DO variable
- The variable of an assignment statement
- The *pointer-object* of a pointer assignment statement
- An *object* or STAT variable in an ALLOCATE or DEALLOCATE statement
- An input item in a READ statement
- A variable name in a NAMELIST statement if the namelist group name appears in a NML specifier in a READ statement
- An internal file unit in a WRITE statement
- A definable variable in an INQUIRE statement
- An IOSTAT or SIZE specifier in an I/O statement
- An actual argument in a reference to a procedure with an explicit interface if the associated dummy argument has intent OUT or INOUT

INTENT on a pointer dummy argument refers to the pointer association status of the pointer and has no effect on the value of the target of the pointer.

A pointer dummy argument with intent IN (or a subobject of such a pointer argument) must *not* appear as any of the following:

- A *pointer-object* in a NULLIFY statement
- A *pointer-object* in a pointer assignment statement
- An *object* in an ALLOCATE or DEALLOCATE statement

- An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

If an actual argument is an array section with a vector subscript, it cannot be associated with a dummy array that is defined or redefined (has intent OUT or INOUT).

Example

The following example shows type declaration statements specifying the INTENT attribute:

```
SUBROUTINE TEST(I, J)
  INTEGER, INTENT(IN) :: I
  INTEGER, INTENT(OUT), DIMENSION(I) :: J
```

The following are examples of the INTENT statement:

```
SUBROUTINE TEST(A, B, X)
   INTENT(INOUT) :: A, B
   ...
SUBROUTINE CHANGE(FROM, TO)
   USE EMPLOYEE_MODULE
   TYPE(EMPLOYEE) FROM, TO
   INTENT(IN) FROM
   INTENT(OUT) TO
   ...
```

The following shows another example:

```
SUBROUTINE AVERAGE(value,data1, cube_ave)
  TYPE DATA
    INTEGER count
    REAL avg
  END TYPE
  TYPE(DATA) data1
  REAL tmp
  ! value cannot be changed, while cube_ave must be defined
  ! before it can be used. Data1 is defined when the procedure is
  ! invoked, and becomes redefined in the subroutine.
  INTENT(IN)::value; INTENT(OUT)::cube_ave
  INTENT(INOUT)::data1
  ! count number of times AVERAGE has been called on the data set
  ! being passed.
  tmp = data1%count*data1%avg + value
  data1%count = data1%count + 1
  data1%avg = tmp/data1%count
  cube_ave = data1%avg**3
END SUBROUTINE
```

See Also

- H to I
- Argument Association
- Type Declarations
- Compatible attributes

INTERFACE

Statement: *Defines explicit interfaces for external or dummy procedures. It can also be used to define a generic name for procedures, a new operator for functions, and a new form of assignment for subroutines.*

Syntax

```
INTERFACE [generic-spec]
   [interface-body]...
   [MODULE PROCEDURE name-list]...
END INTERFACE [generic-spec]
```

generic-spec

(Optional) Is one of the following:

- A generic name

  For information on generic names, see Program Units and Procedures.

- OPERATOR (*op*)

  Defines a generic operator (*op*). It can be a defined unary, defined binary, or extended intrinsic operator. For information on defined operators, see Program Units and Procedures.

- ASSIGNMENT (=)

  Defines generic assignment. For information on defined assignment, see Assignment - Defined Assignment.

interface-body

Is one or more function or subroutine subprograms. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.
The subprogram must *not* contain a statement function or a DATA, ENTRY, or FORMAT statement; an entry name can be used as a procedure name.
The subprogram can contain a USE statement.

name-list	Is the name of one or more module procedures that are accessible in the host. The MODULE PROCEDURE statement is only allowed if the interface block specifies a *generic-spec* and has a host that is a module (or accesses a module by use association). The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprogram definitions.

Description

Interface blocks can appear in the specification part of the program unit that invokes the external or dummy procedure.

A *generic-spec* can only appear in the END INTERFACE statement (a Fortran 95 feature) if one appears in the INTERFACE statement; they must be identical.

The characteristics specified for the external or dummy procedure must be consistent with those specified in the procedure's definition.

An interface block must not appear in a block data program unit.

An interface block comprises its own scoping unit, and does not inherit anything from its host through host association.

Internal, module, and intrinsic procedures are all considered to have explicit interfaces. External procedures have implicit interfaces by default; when you specify an interface block for them, their interface becomes explicit. A procedure must not have more than one explicit interface in a given scoping unit. This means that you cannot include internal, module, or intrinsic procedures in an interface block, unless you want to define a generic name for them.

A interface block containing *generic-spec* specifies a generic interface for the following procedures:

- The procedures within the interface block

  Any generic name, defined operator, or equals symbol that appears is a generic identifier for all the procedures in the interface block. For the rules on how any two procedures with the same generic identifier must differ, see Unambiguous Generic Procedure References.
- The module procedures listed in the MODULE PROCEDURE statement

  The module procedures must be accessible by a USE statement.

To make an interface block available to multiple program units (through a USE statement), place the interface block in a module.

The following rules apply to interface blocks containing pure procedures:

- The interface specification of a pure procedure must declare the INTENT of all dummy arguments except pointer and procedure arguments.
- A procedure that is declared pure in its definition can also be declared pure in an interface block. However, if it is not declared pure in its definition, it must not be declared pure in an interface block.

Example

The following example shows a simple procedure interface block with no generic specification:

```
SUBROUTINE SUB_B (B, FB)
  REAL B
  ...
  INTERFACE
    FUNCTION FB (GN)
      REAL FB, GN
    END FUNCTION
  END INTERFACE
```

The following shows another example:

```
!An interface to an external subroutine SUB1 with header:
!SUBROUTINE SUB1(I1,I2,R1,R2)
!INTEGER I1,I2
!REAL R1,R2
INTERFACE
  SUBROUTINE SUB1(int1,int2,real1,real2)
    INTEGER int1,int2
    REAL real1,real2
  END SUBROUTINE SUB1
END INTERFACE
INTEGER int
. . .
```

See Also

- H to I
- CALL
- FUNCTION
- MODULE
- MODULE PROCEDURE
- SUBROUTINE
- PURE
- Procedure Interfaces
- Use and Host Association
- Determining When Procedures Require Explicit Interfaces
- Defining Generic Names for Procedures
- Defining Generic Operators
- Defining Generic Assignment

INTERFACE TO

Statement: *Identifies a subprogram and its actual arguments before it is referenced or called.*

Syntax

```
INTERFACE TO subprogram-stmt
   [formal-declarations]
END
```

subprogram-stmt	Is a function or subroutine declaration statement.
formal-declarations	(Optional) Are type declaration statements (including optional attributes) for the arguments.

The INTERFACE TO block defines an explicit interface, but it contains specifications for only the procedure declared in the INTERFACE TO statement. The explicit interface is defined only in the program unit that contains the INTERFACE TO statement.

The recommended method for defining explicit interfaces is to use an INTERFACE block.

Example

Consider that a C function that has the following prototype:

```
extern void Foo (int i);
```

The following INTERFACE TO block declares the Fortran call to this function:

```
INTERFACE TO SUBROUTINE Foo [C.ALIAS: '_Foo'] (I)
  INTEGER*4 I
END
```

See Also

- H to I
- INTERFACE

INTRINSIC

Statement and Attribute: *Allows the specific name of an intrinsic procedure to be used as an actual argument.*

Syntax

The INTRINSIC attribute can be specified in a type declaration statement or an INTRINSIC statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] INTRINSIC [, *att-ls*] :: *in-pro*[, *in-pro*]...

Statement:

INTRINSIC [::] *in-pro*[, *in-pro*] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
in-pro	Is the name of an intrinsic procedure.

Description

In a type declaration statement, only *functions* can be declared INTRINSIC. However, you can use the INTRINSIC *statement* to declare subroutines, as well as functions, to be intrinsic.

The name declared INTRINSIC is assumed to be the name of an intrinsic procedure. If a generic intrinsic function name is given the INTRINSIC attribute, the name retains its generic properties.

Some specific intrinsic function names cannot be used as actual arguments. For more information, see Intrinsic Functions Not Allowed as Actual Arguments.

Example

The following example shows a type declaration statement specifying the INTRINSIC attribute:

```
PROGRAM EXAMPLE
...
REAL(8), INTRINSIC :: DACOS
...
CALL TEST(X, DACOS)     ! Intrinsic function DACOS is an actual argument
```

The following example shows an INTRINSIC statement:

Main Program	Subprogram
EXTERNAL CTN	SUBROUTINE TRIG(X,F,Y)
INTRINSIC SIN, COS	Y = F(X)
...	RETURN
	END
CALL TRIG(ANGLE,SIN,SINE)	
...	FUNCTION CTN(X)
	CTN = COS(X)/SIN(X)
CALL TRIG(ANGLE,COS,COSINE)	RETURN
...	END
CALL TRIG(ANGLE,CTN,COTANGENT)	

Note that when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the Fortran 95/90 library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

The following shows another example:

```
     INTRINSIC SIN, COS
     REAL X, Y, R
 !   SIN and COS are arguments to Calc2:
     R = Calc2 (SIN, COS)
```

See Also

- H to I
- References to Generic Procedures
- Type Declarations
- Compatible attributes

INUM

Elemental Intrinsic Function (Specific):
Converts a character string to an INTEGER(2) value. This function cannot be passed as an actual argument.

Syntax

```
result = INUM (i)
```

i (Input) Must be of type character.

Results

The result type is INTEGER(2). The result value is the INTEGER(2) value represented by the character string *i*.

Example

INUM ("451") has the value 451 of type INTEGER(2).

IOR

Elemental Intrinsic Function (Generic):
Performs an inclusive OR on corresponding bits. This function can also be specified as OR.

Syntax

```
result = IOR (i,j)
```

i	(Input) Must be of type integer or of type logical (which is treated as an integer).
j	(Input) Must be of type integer with the same kind parameter as *i*. If the kinds of *i* and *j* do not match, the value with the smaller kind is extended with zeros on the left and the larger kind is used for the operation and the result.

Results

The result type is the same as *i*. The result value is derived by combining *i* and *j* bit-by-bit according to the following truth table:

i	j	IOR (i, j)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BIOR	INTEGER(1)	INTEGER(1)
IIOR[1]	INTEGER(2)	INTEGER(2)
JIOR	INTEGER(4)	INTEGER(4)
KIOR	INTEGER(8)	INTEGER(8)

[1] Or HIOR.

Example

IOR (1, 4) has the value 5.

IOR (1, 2) has the value 3.

The following shows another example:

```
INTEGER result

result = IOR(240, 90) ! returns 250
```

See Also

- H to I
- IAND
- IEOR
- NOT

IPXFARGC

POSIX Function: *Returns the index of the last command-line argument.*

Module

USE IFPOSIX

Syntax

```
result = IPXFARGC( )
```

Results

The result type is INTEGER(4). The result value is the number of command-line arguments, excluding the command name, in the command used to invoke the executing program. A return value of zero indicates there are no command-line arguments other than the command name itself.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- PXFGETARG

IPXFCONST

POSIX Function: *Returns the value associated with a constant defined in the C POSIX standard.*

Module

USE IFPOSIX

Syntax

```
result = IPXFCONST (constname)
```

constname (Input) Character. The name of a C POSIX standard constant.

Results

The result type is INTEGER(4). If *constname* corresponds to a defined constant in the C POSIX standard, the result value is the integer that is associated with the constant. Otherwise, the result value is -1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- PXFGETARG
- PXFCONST

IPXFLENTRIM

POSIX Function: *Returns the index of the last non-blank character in an input string.*

Module

USE IFPOSIX

Syntax

```
result = IPXFLENTRIM (string)
```

string (Input) Character. A character string.

Results

The result type is INTEGER(4). The result value is the index of the last non-blank character in the input argument *string*, or zero if all characters in *string* are blank characters.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

IPXFWEXITSTATUS (L*X, M*X)

POSIX Function: *Returns the exit code of a child process.*

Module

USE IFPOSIX

Syntax

```
result = IPXFWEXITSTATUS (istat)
```

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the low-order eight bits of the output argument of PXFWAIT or PXFWAITPID.

The IPXFWEXITSTATUS function should only be used if PXFWIFEXITED returns TRUE.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
  end if
10 FORMAT (A,Z)
```

```
end program
```

See Also

- H to I
- PXFWAIT
- PXFWAITPID
- PXFWIFEXITED

IPXFWSTOPSIG (L*X, M*X)

POSIX Function: *Returns the number of the signal that caused a child process to stop.*

Module

USE IFPOSIX

Syntax

`result = IPXFWSTOPSIG (istat)`

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the number of the signal that caused the child process to stop.

The IPXFWSTOPSIG function should only be used if PXFWIFSTOPPED returns TRUE.

See Also

- H to I
- PXFWAIT
- PXFWAITPID
- PXFWIFSTOPPED

IPXFWTERMSIG (L*X, M*X)

POSIX Function: *Returns the number of the signal that caused a child process to terminate.*

Module

USE IFPOSIX

Syntax

```
result = IPXFWTERMSIG (istat)
```

istat	(Input) INTEGER(4). The value of output argument *istat* from PXFWAIT or PXFWAITPID.

Results

The result type is INTEGER(4). The result is the number of the signal that caused the child process to terminate.

The IPXFWTERMSIG function should only be used if PXFWIFSIGNALED returns TRUE.

See Also

- H to I
- PXFWAIT
- PXFWAITPID
- PXFWIFSIGNALED

IRAND, IRANDM

Portability Functions: *Return random numbers in the range 0 through (2**31)-1, or 0 through (2**15)-1 if called without an argument.*

Module

USE IFPORT

Syntax

```
result = IRAND ([iflag])
```

```
result = IRANDM ([iflag])
```

iflag	(Input) INTEGER(4). Optional for IRAND. Controls the way the returned random number is chosen. If *iflag* is omitted, it is assumed to be 0, and the return range is 0 through (2**15)-1 (inclusive).

Results

The result type is INTEGER(4). If *iflag* is 1, the generator is restarted and the first random value is returned. If *iflag* is 0, the next random number in the sequence is returned. If *iflag* is neither zero nor 1, it is used as a new seed for the random number generator, and the functions return the first new random value.

IRAND and IRANDM are equivalent and return the same random numbers. Both functions are included to ensure portability of existing code that references one or both of them.

You can use SRAND to restart the pseudorandom number generator used by these functions.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) istat, flag_value, r_nums(20)
flag_value=1
r_nums(1) = IRAND (flag_value)
flag_value=0
do istat=2,20
   r_nums(istat) = irand(flag_value)
end do
```

See Also

- H to I
- RANDOM_NUMBER
- RANDOM_SEED
- SRAND

Building Applications: Portability Library Overview

IRANGET

Portability Subroutine: *Returns the current seed.*

Module

USE IFPORT

Syntax

```
CALL IRANGET (seed)
```

seed	(Output) INTEGER(4). The current seed value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- IRANSET

IRANSET

Portability Subroutine: *Sets the seed for the random number generator.*

Module

USE IFPORT

Syntax

```
CALL IRANSET (seed)
```

seed	(Input) INTEGER(4). The reset value for the seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- H to I
- IRANGET

ISATTY

Portability Function: *Checks whether a logical unit number is a terminal.*

Module

USE IFPORT

Syntax

`result = ISATTY (lunit)`

iunit	(Input) INTEGER(4). An integer expression corresponding to a Fortran logical unit number. Must be in the range 0 to 100 and must be connected.

Results

The result type is LOGICAL(4). The result is .TRUE. if the specified logical unit is connected to a terminal device; otherwise, .FALSE..

If *lunit* is out of range or is not connected, zero is returned.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

IS_IOSTAT_END

Elemental Intrinsic Function (Generic): *Tests for an end-of-file condition.*

Syntax

`result=IS_IOSTAT_END(i)`

i	(Input) Must be of type integer.

Results

The result type is default logical. The value of the result is true only if *i* is a value that could be assigned to the scalar integer variable in an IOSTAT= specifier to indicate an end-of-file condition.

Example

```
INTEGER IO_STATUS
…
READ (20, IOSTAT=IO_STATUS) A, B, C
IF (IS_IOSTAT_END (IO_STATUS)) THEN
…                                ! process end of file
ENDIF
…                                ! process data read
```

IS_IOSTAT_EOR

Elemental Intrinsic Function (Generic): *Tests for an end-of-record condition.*

Syntax

`result=IS_IOSTAT_EOR(i)`

i (Input) Must be of type integer.

Results

The result type is default logical. The value of the result is true only if *i* is a value that could be assigned to the scalar integer variable in an IOSTAT= specifier to indicate an end-of-record condition.

Example

```
INTEGER IO_STATUS
…
READ (30, ADVANCE='YES', IOSTAT=IO_STATUS) A, B, C
IF (IS_IOSTAT_EOR (IO_STATUS)) THEN
…                                ! process end of record
ENDIF
…                                ! process data read
```

ISHA

Elemental Intrinsic Function (Generic):
Arithmetically shifts an integer left or right by a specified number of bits.

Syntax

`result = ISHA (i,shift)`

i	(Input) Must be of type integer. This argument is the value to be shifted.
shift	(Input) Must be of type integer. This argument is the direction and distance of shift. Positive shifts are left (toward the most significant bit); negative shifts are right (toward the least significant bit).

Results

The result type is the same as *i*. The result is equal to *i* shifted arithmetically by *shift* bits.

If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. If the shift is to the left, zeros are shifted in on the right. If the shift is to the right, copies of the sign bit (0 for non-negative *i*; 1 for negative *i*) are shifted in on the left.

The kind of integer is important in arithmetic shifting because sign varies among integer representations (see the following example). If you want to shift a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = -128             ! equal to 10000000
j = -32768           ! equal to 10000000 00000000
res1 = ISHA (i, -4)  ! returns 11111000 = -8
res2 = ISHA (j, -4)  ! returns 11111000 10100000 = -2048
```

See Also

- H to I
- ISHC
- ISHL
- ISHFT
- ISHFTC

ISHC

Elemental Intrinsic Function (Generic): *Rotates an integer left or right by specified number of bits. Bits shifted out one end are shifted in the other end. No bits are lost.*

Syntax

```
result = ISHC (i,shift)
```

i	(Input) Must be of type integer. This argument is the value to be rotated.
shift	(Input) Must be of type integer. This argument is the direction and distance of rotation. Positive rotations are left (toward the most significant bit); negative rotations are right (toward the least significant bit).

Results

The result type is the same as *i*. The result is equal to *i* circularly rotated by *shift* bits.

If *shift* is positive, *i* is rotated left *shift* bits. If *shift* is negative, *i* is rotated right *shift* bits. Bits shifted out one end are shifted in the other. No bits are lost.

The kind of integer is important in circular shifting. With an INTEGER(4) argument, all 32 bits are shifted. If you want to rotate a one-byte or two-byte argument, you must declare it as INTEGER(1) or INTEGER(2).

Example

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10  ! equal to 00001010
j = 10  ! equal to 00000000 00001010
res1 = ISHC (i, -3)   ! returns 01000001 = 65
res2 = ISHC (j, -3)   ! returns 01000000 00000001 =
                      ! 16385
```

See Also

- H to I
- ISHA
- ISHL
- ISHFT
- ISHFTC

ISHFT

Elemental Intrinsic Function (Generic):
Performs a logical shift.

Syntax

`result = ISHFT (i,shift)`

i	(Input) Must be of type integer.
shift	(Input) Must be of type integer. The absolute value for *shift* must be less than or equal to BIT_SIZE(*i*).

Results

The result type is the same as *i*. The result has the value obtained by shifting the bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

ISHFT with a positive *shift* can also be specified as LSHIFT (or LSHFT). ISHFT with a negative *shift* can also be specified as RSHIFT (or RSHFT) with | *shift* |.

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BSHFT	INTEGER(1)	INTEGER(1)
IISHFT[1]	INTEGER(2)	INTEGER(2)
JISHFT	INTEGER(4)	INTEGER(4)
KISHFT	INTEGER(8)	INTEGER(8)

[1]Or HSHFT.

Example

ISHFT (2, 1) has the value 4.

ISHFT (2, -1) has the value 1.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, k(3), res2
i = 10 ! equal to 00001010
j = 10 ! equal to 00000000 00001010
res1 = ISHFT (i, 5)  ! returns 01000000 = 64
res2 = ISHFT (j, 5)  ! returns 00000001 01000000 =
                     ! 320
k = ISHFT((/3, 5, 1/), (/1, -1, 0/))  ! returns array
                                      ! /6, 2, 1/
```

See Also

- H to I

- BIT_SIZE
- ISHFTC
- ISHA
- ISHC

ISHFTC

Elemental Intrinsic Function (Generic):
Performs a circular shift of the rightmost bits.

Syntax

```
result = ISHFTC (i,shift[,size])
```

i	(Input) Must be of type integer.
shift	(Input) Must be of type integer. The absolute value of *shift* must be less than or equal to *size*.
size	(Input;optional) Must be of type integer. The value of *size*must be positive and must not exceed BIT_SIZE(*i*). If *size*is omitted, it is assumed to have the value of BIT_SIZE(*i*).

Results

The result type is the same as *i*. The result value is obtained by circular shifting the *size* rightmost bits of *i* by *shift* positions. If *shift* is positive, the shift is to the left; if *shift* is negative, the shift is to the right. If *shift* is zero, no shift is performed.

No bits are lost. Bits in *i* beyond the value specified by *size* are unaffected.

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BSHFTC	INTEGER(1)	INTEGER(1)
IISHFTC[1]	INTEGER(2)	INTEGER(2)
JISHFTC	INTEGER(4)	INTEGER(4)
KISHFTC	INTEGER(8)	INTEGER(8)

Specific Name	Argument Type	Result Type
[1]Or HSHFTC.		

Example

ISHFTC (4, 2, 4) has the value 1.

ISHFTC (3, 1, 3) has the value 6.

The following shows another example:

```
INTEGER(1) i, res1
INTEGER(2) j, res2
i = 10  ! equal to 00001010
j = 10  ! equal to 00000000 00001010
res1 = ISHFTC (i, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 (left) and
                          ! returns 00001001 = 9
res1 = ISHFTC (i, -2, 3)  ! rotates the 3 rightmost
                          ! bits by -2 (right) and
                          ! returns 00001100 = 12
res2 = ISHFTC (j, 2, 3)   ! rotates the 3 rightmost
                          ! bits by 2 and returns
                          ! 00000000 00001001 = 9
```

See Also

- H to I
- BIT_SIZE
- ISHFT
- MVBITS

ISHL

Elemental Intrinsic Function (Generic):
Logically shifts an integer left or right by the specified bits. Zeros are shifted in from the opposite end.

Syntax

`result = ISHL (i,shift)`

i	(Input) Must be of type integer. This argument is the value to be shifted.
shift	(Input) Must be of type integer. This argument is the direction and distance of shift. If positive, *i* is shifted left (toward the most significant bit). If negative, *i* is shifted right (toward the least significant bit).

Results

The result type is the same as *i*. The result is equal to *i* logically shifted by *shift* bits. Zeros are shifted in from the opposite end.

Unlike circular or arithmetic shifts, which can shift ones into the number being shifted, logical shifts shift in zeros only, regardless of the direction or size of the shift. The integer kind, however, still determines the end that bits are shifted out of, which can make a difference in the result (see the following example).

Example

```
INTEGER(1) i, res1

INTEGER(2) j, res2

i = 10   ! equal to 00001010

j = 10   ! equal to 00000000 00001010

res1 = ISHL (i, 5)   ! returns 01000000 = 64

res2 = ISHL (j, 5)   ! returns 00000001 01000000 = 320
```

See Also

- H to I
- ISHA

- ISHC
- ISHFT
- ISHFTC

ISNAN

Elemental Intrinsic Function (Generic): *Tests whether IEEE* real (S_floating, T_floating, and X_floating) numbers are Not-a-Number (NaN) values.*

Syntax

```
result = ISNAN (x)
```

x (Input) Must be of type real.

Results

The result type is default logical. The result is .TRUE. if *x* is an IEEE NaN; otherwise, the result is .FALSE..

Example

```
LOGICAL A
DOUBLE PRECISION B
...
A = ISNAN(B)
```

A is assigned the value .TRUE. if B is an IEEE NaN; otherwise, the value assigned is .FALSE..

ITIME

Portability Subroutine: *Returns the time in numeric form.*

Module

USE IFPORT

Syntax

```
CALL ITIME (array)
```

array	(Output) INTEGER(4). A rank one array with three elements used to store numeric time data: • *array*(1) - the hour • *array*(2) - the minute • *array*(3) - the second

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
 INTEGER(4) time_array(3)
 CALL ITIME (time_array)
 write(*,10) time_array
 10 format (1X,I2,':',I2,':',I2)
 END
```

See Also

- H to I
- DATE_AND_TIME

Building Applications: Portability Library Overview

IVDEP

General Compiler Directive: *Assists the compiler's dependence analysis of iterative DO loops.*

Syntax

```
cDEC$ IVDEP [: option]
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
option	Is LOOP or BACK. This argument is only available on processors using IA-32 architecture.

Description

The IVDEP directive is an assertion to the compiler's optimizer about the order of memory references inside a DO loop.

IVDEP:LOOP implies no loop-carried dependencies. IVDEP:BACK implies no backward dependencies.

When no *option* is specified, the following occurs:

- On IA-64 architecture, the behavior is the same as IVDEP:BACK. You can modify the behavior to be the same as IVDEP:LOOP by using a compiler option.
- On Intel® 64 architecture and IA-32 architecture, the compiler begins dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.

*c*DEC$ IVDEP with no option can also be spelled *c*DEC$ INIT_DEP_FWD (INITialize DEPendences ForWarD).

The IVDEP directive is applied to a DO loop in which the user knows that dependences are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.

The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependences, to choose other execution orders.

Example

In the following example, the IVDEP directive provides more information about the dependences within the loop, which may enable loop transformations to occur:

```
!DEC$ IVDEP
      DO I=1, N
         A(INDARR(I)) = A(INDARR(I)) + B(I)
      END DO
```

In this case, the scalar execution order follows:

1. Retrieve INDARR(I).
2. Use the result from step 1 to retrieve A(INDARR(I)).
3. Retrieve B(I).
4. Add the results from steps 2 and 3.
5. Store the results from step 4 into the location indicated by A(INDARR(I)) from step 1.

IVDEP directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.

See Also

- H to I
- General Compiler Directives
- Rules for General Directives that Affect DO Loops

IXOR

Elemental Intrinsic Function (Generic): *See IEOR.*

J to L

JABS

Portability Function: *Returns an absolute value.*

Module

USE IFPORT

Syntax

```
result = JABS (i)
```

i (Input) INTEGER(4). A value.

Results

The result type is INTEGER(4). The value of the result is | *i* |.

JDATE

Portability Function: *Returns an 8-character string with the Julian date in the form "yyddd". Three spaces terminate this string.*

Module

USE IFPORT

Syntax

```
result = JDATE( )
```

Results

The result type is character with length 8. The result is the Julian date, in the form YYDDD, followed by three spaces.

The Julian date is a five-digit number whose first two digits are the last two digits of the year, and whose final three digits represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on). For example, the Julian date for February 1, 1999 is 99032.

CAUTION. The two-digit year return value may cause problems with the year 2000. Use DATE_AND_TIMEinstead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
! Sets julian to today's julian date
    USE IFPORT
    CHARACTER*8 julian
    julian = JDATE( )
```

See Also

- J to L
- DATE_AND_TIME

JDATE4

Portability Function: *Returns a 10-character string with the Julian date in the form "yyyyddd". Three spaces terminate this string.*

Module

USE IFPORT

Syntax

```
result = JDATE4( )
```

Results

The result type is character with length 10. The result is the Julian date, in the form YYYYDDD, followed by three spaces.

The Julian date is a seven-digit number whose first four digits are the year, and whose final three represent the day of the year (1 for January 1, 366 for December 31 of a leap year, and so on). For example, the Julian date for February 1, 1999 is 1999032.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- J to L
- DATE_AND_TIME

JNUM

Elemental Intrinsic Function (Specific): *Converts a character string to an INTEGER(4) value. This function cannot be passed as an actual argument.*

Syntax

```
result = JNUM (i)
```

i (Input) Must be of type character.

Results

The result type is INTEGER(4). The result value is the integer value represented by the character string *i*.

Example

JNUM ("46616") has the value 46616 of type INTEGER(4).

KILL

Portability Function: *Sends a signal to the process given by ID.*

Module

USE IFPORT

Syntax

`result = KILL (pid,signum)`

pid	(Input) INTEGER(4). ID of a process to be signaled.
signum	(Input) INTEGER(4). A signal value. For the definition of signal values, see the SIGNALfunction.

Results

The result type is INTEGER(4). The result is zero if the call was successful; otherwise, an error code. Possible error codes are:

- EINVAL: The *signum* is not a valid signal number, or PID is not the same as getpid( ) and *signum* does not equal SIGKILL.
- ESRCH: The given PID could not be found.
- EPERM: The current process does not have permission to send a signal to the process given by PID.

On Windows* systems, arbitrary signals can be sent only to the calling process (where *pid*= getpid( )). Other processes can send only the SIGKILL signal (*signum*= 9), and only if the calling process has permission.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

integer(4) id_number, sig_val, istat

id_number=getpid( )

ISTAT = KILL (id_number, sig_val)
```

See Also

- J to L
- SIGNAL
- RAISEQQ
- SIGNALQQ

Building Applications: Portability Library Overview

KIND

Inquiry Intrinsic Function (Generic): *Returns the kind parameter of the argument.*

Syntax

```
result = KIND (x)
```

x	(Input) Can be of any intrinsic type.

Results

The result is a scalar of type default integer. The result has a value equal to the kind type parameter value of *x*.

Example

KIND (0.0) has the kind value of default real type.

KIND (12) has the kind value of default integer type.

The following shows another example:

```
INTEGER i ! a 4-byte integer
WRITE(*,*) KIND(i)
CALL INTEGER2( )
WRITE(*,*) KIND(i)  ! still a 4-byte integer
                    ! not affected by setting in subroutine
END
SUBROUTINE INTEGER2( )
  !DEC$INTEGER:2
  INTEGER j ! a 2-byte integer
  WRITE(*,*) KIND(j)
END SUBROUTINE
```

See Also

- J to L
- SELECTED_INT_KIND
- SELECTED_REAL_KIND
- CMPLX
- INT
- REAL
- LOGICAL
- CHAR
- Intrinsic Data Types
- Argument Keywords in Intrinsic Procedures

KNUM

Elemental Intrinsic Function (Specific):
Converts a character string to an INTEGER(8) value.

Syntax

`result = KNUM (i)`

i	(Input) Must be of type character.

Results

The result type is INTEGER(8). The result value is the integer value represented by the character string *i*.

Example

KNUM ("46616") has the value 46616 of type INTEGER(8).

LASTPRIVATE

Parallel Directive Clause: *Provides a superset of the functionality provided by the PRIVATE clause; objects are declared PRIVATE and they are given certain values when the parallel region is exited.*

Syntax

```
LASTPRIVATE (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, once the parallel region is exited, each variable has the value provided by the sequentially last section or loop iteration.

When the LASTPRIVATE clause appears in a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct.

Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

LBOUND

Inquiry Intrinsic Function (Generic): *Returns the lower bounds for all dimensions of an array, or the lower bound for a specified dimension.*

Syntax

```
result = LBOUND (array [, dim] [, kind])
```

array	(Input) Must be an array. It may be of any data type. It must not be an allocatable array that is not allocated, or a disassociated pointer.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank *array*.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is a scalar. Otherwise, the result is a rank-one array with one element for each dimension of *array*. Each element in the result corresponds to a dimension of *array*.

If *array* is an array section or an array expression that is not a whole array or array structure component, each element of the result has the value 1.

If *array* is a whole array or array structure component, LBOUND (*array*, *dim*) has a value equal to the lower bound for subscript *dim* of *array*(if *dim* is nonzero or *array* is an assumed-size array of rank *dim*). Otherwise, the corresponding element of the result has the value 1.

The setting of compiler options specifying integer size can affect this function.

Example

Consider the following:

```
REAL ARRAY_A (1:3, 5:8)

REAL ARRAY_B (2:8, -3:20)
```

LBOUND(ARRAY_A) is (1, 5). LBOUND(ARRAY_A, DIM=2) is 5.

LBOUND(ARRAY_B) is (2, -3). LBOUND(ARRAY_B (5:8, :)) is (1,1) because the arguments are array sections.

The following shows another example:

```
REAL ar1(2:3, 4:5, -1:14), vec1(35)

INTEGER res1(3), res2, res3(1)

res1 = UBOUND (ar1)          ! returns [2, 4, -1]

res2 = UBOUND (ar1, DIM= 3)  ! returns -1

res3 = UBOUND (vec1)         ! returns [1]

END
```

See Also

- J to L
- UBOUND

LCWRQQ

Portability Subroutine: *Sets the value of the floating-point processor control word.*

Module

USE IFPORT

Syntax

CALL LCWRQQ (*controlword*)

controlword (Input) INTEGER(2). Floating-point processor control word.

LCWRQQ performs the same function as the run-time subroutine SETCONTROLFPQQ and is provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(2) control
CALL SCWRQQ(control) ! get control word
!  Set control word to make processor round up
control = control .AND. (.NOT. FPCW$MCW_RC)  ! Clear
                                             !  control word with inverse
                                             !  of rounding control mask
control = control .OR. FPCW$UP  ! Set control word
                                !  to round up
CALL LCWRQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

See Also

- J to L
- SETCONTROLFPQQ

LEADZ

Elemental Intrinsic Function (Generic):
Returns the number of leading zero bits in an integer.

Syntax

`result = LEADZ (i)`

i (Input) Must be of type integer or logical.

Results

The result type is the same as *i*. The result value is the number of leading zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

Consider the following:

```
INTEGER*8 J, TWO

PARAMETER (TWO=2)

DO J= -1, 40

  TYPE *, LEADZ(TWO**J)  ! Prints 64 down to 23 (leading zeros)

ENDDO

END
```

LEN

Inquiry Intrinsic Function (Generic): *Returns the length of a character expression.*

Syntax

`result = LEN (`*`string`* `[, `*`kind`*`])`

`string`	(Input) Must be of type character; it can be scalar or array valued. (It can be an array of strings.)
`kind`	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is a scalar of type integer. If *`kind`* is present, the kind parameter of the result is that specified by *`kind`*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters in *`string`*(if it is scalar) or in an element of *`string`*(if it is array valued).

Specific Name	Argument Type	Result Type
LEN [1]	CHARACTER	INTEGER(4)
	CHARACTER	INTEGER(8)

Specific Name	Argument Type	Result Type

[1]The setting of compiler options specifying integer size can affect this function.

Example

Consider the following example:

```
CHARACTER (15) C (50)
CHARACTER (25) D
```

LEN (C) has the value 15, and LEN (D) has the value 25.

The following shows another example:

```
CHARACTER(11) STR(100)
INTEGER I
I = LEN (STR)  ! returns 11
I = LEN('A phrase with 5 trailing blanks. ')
               ! returns 37
```

See Also

- J to L
- LEN_TRIM
- Declaration Statements for Character Types
- Character Data Type

LEN_TRIM

Elemental Intrinsic Function (Generic):
Returns the length of the character argument without counting trailing blank characters.

Syntax

```
result = LEN_TRIM (string [, kind])
```

string	(Input) Must be of type character.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result has a value equal to the number of characters remaining after any trailing blanks in *string* are removed. If the argument contains only blank characters, the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

LEN_TRIM (' C D ') has the value 4.

LEN_TRIM (' ') has the value 0.

The following shows another example:

```
INTEGER LEN1

LEN1 = LEN_TRIM (' GOOD DAY ') ! returns 9

LEN1 = LEN_TRIM (' ')          ! returns 0
```

See Also

- J to L
- LEN
- LNBLNK

LGE

Elemental Intrinsic Function (Generic):
Determines if a string is lexically greater than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LGE is equivalent to the (.GE.) operator.

Syntax

```
result = LGE (string_a,string_b)
```

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LGE [1,2]	CHARACTER	LOGICAL(4)

[1] This specific function cannot be passed as an actual argument.

[2] The setting of compiler options specifying integer size can affect this function.

Example

LGE ('ONE', 'SIX') has the value false.

LGE ('TWO', 'THREE') has the value true.

The following shows another example:

```
LOGICAL L

L = LGE('ABC','ABD')       ! returns .FALSE.

L = LGE ('AB', 'AAAAAAAB') ! returns .TRUE.
```

See Also

- J to L
- LGT
- LLE
- LLT
- ASCII and Key Code Charts

LGT

Elemental Intrinsic Function (Generic):
Determines whether a string is lexically greater than another string, based on the ASCII collating

sequence, even if the processor's default collating sequence is different. In Intel© Fortran, LGT is equivalent to the > (.GT.) operator.

Syntax

`result = LGT (string_a,string_b)`

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* follows *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LGT [1,2]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

[2] The setting of compiler options specifying integer size can affect this function.

Example

LGT ('TWO', 'THREE') has the value true.

LGT ('ONE', 'FOUR') has the value true.

The following shows another example:

```
LOGICAL L

L = LGT('ABC', 'ABC')  ! returns .FALSE.

L = LGT('ABC', 'AABC') ! returns .TRUE.
```

See Also

- J to L
- LGE
- LLE

- LLT
- ASCII and Key Code Charts

LINETO, LINETO_W (W*32, W*64)

Graphics Function: *Draws a line from the current graphics position up to and including the end point.*

Module

USE IFQWIN

Syntax

```
result = LINETO (x,y)
result = LINETO_W (wx, wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates of end point.
wx, wy	(Input) REAL(8). Window coordinates of end point.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, 0.

The line is drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

If no error occurs, LINETO sets the current graphics position to the viewport point (*x*, *y*), and LINETO_W sets the current graphics position to the window point (*wx*, *wy*).

If you use FLOODFILLRGB to fill in a closed figure drawn with LINETO, the figure must be drawn with a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE. The LINETO routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the LineTo routine by including the IFWIN module, you need to specify the routine name as MSFWIN$LineTo. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

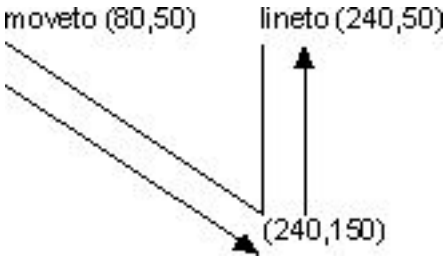
Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

Example

This program draws the figure shown below.

```
! Build as QuickWin or Standard Graphics
USE IFQWIN
INTEGER(2) status
TYPE (xycoord) xy
CALL MOVETO(INT2(80), INT2(50), xy)
status = LINETO(INT2(240), INT2(150))
status = LINETO(INT2(240), INT2(50))
END
```



See Also

- J to L
- GETCURRENTPOSITION
- GETLINESTYLE
- GRSTATUS
- MOVETO
- POLYGON
- POLYLINEQQ
- SETLINESTYLE
- SETWRITEMODE

Building Applications: Drawing Lines on the Screen

Building Applications: Graphics Coordinates

LINETOAR (W*32, W*64)

Graphics Function: *Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array.*

Module

USE IFQWIN

Syntax

```
result = LINETOAR (loc(fx), loc(fy), loc(tx) loc(ty), cnt)
```

fx	(Input) INTEGER(2). From x viewport coordinate array.
fy	(Input) INTEGER(2). From y viewport coordinate array.
tx	(Input) INTEGER(2). To x viewport coordinate array.
ty	(Input) INTEGER(2). To y viewport coordinate array.
cnt	(Input) INTEGER(4). Length of each coordinate array; all should be the same size.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) cnt, i
! load the points
do i = 1,3
  !from here
  fx(i) =20*i
  fy(i) =10
  !to there
  tx(i) =20*i
  ty(i) =60
end do
! draw the lines all at once
! 3 white vertical lines in upper left corner
result = LINETOAR(loc(fx),loc(fy),loc(tx),loc(ty), 3)
end
```

See Also

- J to L
- LINETO
- LINETOAREX
- LOC
- SETCOLORRGB
- SETLINESTYLE
- SETWRITEMODE

Building Applications: Drawing Lines on the Screen

LINETOAREX (W*32, W*64)

Graphics Function: *Draws a line between each x,y point in the from-array to each corresponding x,y point in the to-array. Each line is drawn with the specified graphics color and line style.*

Module

USE IFQWIN

Syntax

```
result = LINETOAREX (loc(fx), loc(fy), loc(tx) loc(ty), loc(C), loc(S), cnt)
```

fx	(Input) INTEGER(2). From x viewport coordinate array.
fy	(Input) INTEGER(2). From y viewport coordinate array.
tx	(Input) INTEGER(2). To x viewport coordinate array.
ty	(Input) INTEGER(2). To y viewport coordinate array.
C	(Input) INTEGER(4). Color array.
S	(Input) INTEGER(4). Style array.
cnt	(Input) INTEGER(4). Length of each coordinate array; also the length of the color array and style array. All of the arrays should be the same size.

Results

The result type is INTEGER(2). The result is a nonzero value if successful; otherwise, zero.

The lines are drawn using the specified graphics colors and line styles, and with the current write mode. The current write mode is set with SETWRITEMODE.

If the color has the Z'80000000' bit set, the color is an RGB color; otherwise, the color is a palette color.

The styles are as follows from `wingdi.h`:

```
SOLID           0
DASH            1       /* -------  */
DOT             2       /* .......  */
DASHDOT         3       /* _._._._  */
DASHDOTDOT      4       /* _.._.._  */
NULL            5
```

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
integer(2) fx(3),fy(3),tx(3),ty(3),result
integer(4) C(3),S(3),cnt,i,color
color = #000000FF
! load the points
do i = 1,3
   S(i) = 0  ! all lines solid
   C(i) = IOR(Z'80000000',color)
   color = color*256 ! pick another of RGB
   if(IAND(color,Z'00FFFFFF').eq.0) color = Z'000000FF'
   !from here
   fx(i) =20*i
   fy(i) =10
   !to there
   tx(i) =20*i
   ty(i) =60
 end do
 ! draw the lines all at once
 ! 3 vertical lines in upper left corner, Red, Green, and Blue
 result = LINETOAREX(loc(fx),loc(fy),loc(tx),loc(ty),loc(C),loc(S),3)
 end
```

See Also

- J to L
- LINETO
- LINETOAR
- LOC
- POLYLINEQQ

- SETWRITEMODE

Building Applications: Drawing Lines on the Screen

LLE

Elemental Intrinsic Function (Generic):
Determines whether a string is lexically less than or equal to another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel© Fortran, LLE is equivalent to the (.LE.) operator.

Syntax

```
result = LLE (string_a,string_b)
```

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if the strings are equal, both strings are of zero length, or if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false.

Specific Name	Argument Type	Result Type
LLE [1,2]	CHARACTER	LOGICAL(4)

[1]This specific function cannot be passed as an actual argument.

[2] The setting of compiler options specifying integer size can affect this function.

Example

LLE ('TWO', 'THREE') has the value false.

LLE ('ONE', 'FOUR') has the value false.

The following shows another example:

```
LOGICAL L

L = LLE('ABC', 'ABC')   ! returns .TRUE.

L = LLE('ABC', 'AABCD') ! returns .FALSE.
```

See Also

- J to L
- LGE
- LGT
- LLT
- ASCII and Key Code Charts

LLT

Elemental Intrinsic Function (Generic):
Determines whether a string is lexically less than another string, based on the ASCII collating sequence, even if the processor's default collating sequence is different. In Intel® Fortran, LLT is equivalent to the < (.LT.) operator.

Syntax

`result = LLT (string_a,string_b)`

string_a	(Input) Must be of type character.
string_b	(Input) Must be of type character.

Results

The result type is default logical. If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks, to the length of the longer string.

The result is true if *string_a* precedes *string_b* in the ASCII collating sequence; otherwise, the result is false. If both strings are of zero length, the result is also false.

Specific Name	Argument Type	Result Type
LLT [1,2]	CHARACTER	LOGICAL(4)

Specific Name	Argument Type	Result Type

[1]This specific function cannot be passed as an actual argument.

[2] The setting of compiler options specifying integer size can affect this function.

Example

LLT ('ONE', 'SIX') has the value true.

LLT ('ONE', 'FOUR') has the value false.

The following shows another example:

```
LOGICAL L
L = LLT ('ABC', 'ABC')     ! returns .FALSE.
L = LLT ('AAXYZ', 'ABCDE') ! returns .TRUE.
```

See Also

- J to L
- LGE
- LGT
- LLE
- ASCII and Key Code Charts

LNBLNK

Portability Function: *Locates the position of the last nonblank character in a string.*

Module

USE IFPORT

Syntax

`result = LNBLNK (string)`

string (Input) Character*(*). String to be searched. Cannot be an array.

Results

The result type is INTEGER(4). The result is the index of the last nonblank character in *string*.

LNBLNK is very similar to the intrinsic function LEN_TRIM, except that *string* cannot be an array.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

integer(4) p

p = LNBLNK(' GOOD DAY ') ! returns 9

p = LNBLNK(' ')          ! returns 0
```

See Also

- J to L
- LEN_TRIM

Building Applications: Portability Library Overview

LOADIMAGE, LOADIMAGE_W (W*32, W*64)

Graphics Functions: *Read an image from a Windows bitmap file and display it at a specified location.*

Module

USE IFQWIN

Syntax

`result = LOADIMAGE (filename,xcoord,ycoord)`

`result = LOADIMAGE_W (filename,wxcoord,wycoord)`

filename	(Input) Character*(*). Path of the bitmap file.
xcoord, ycoord	(Input) INTEGER(4). Viewport coordinates for upper-left corner of image display.
wxcoord, wycoord	(Input) REAL(8). Window coordinates for upper-left corner of image display.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The image is displayed with the colors in the bitmap file. If the color palette in the bitmap file is different from the current system palette, the current palette is discarded and the bitmap's palette is loaded.

LOADIMAGE specifies the screen placement of the image in viewport coordinates. LOADIMAGE_W specifies the screen placement of the image in window coordinates.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- J to L
- SAVEIMAGE, SAVEIMAGE_W

Building Applications: Loading and Saving Images to Files

LOC

Inquiry Intrinsic Function (Generic): *Returns the internal address of a storage item. This function cannot be passed as an actual argument.*

Syntax

```
result = LOC (x)
```

x	(Input) Is a variable, an array or record field reference, a procedure, or a constant; it can be of any data type. It must not be the name of a statement function. If it is a pointer, it must be defined and associated with a target.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The value of the result represents the address of the data object or, in the case of pointers, the address of its associated target. If the argument is not valid, the result is undefined.

This function performs the same function as the %LOC built-in function.

Example

```
! Mixed language example passing Integer Pointer to C
! Fortran main program
      INTERFACE
         SUBROUTINE Ptr_Sub (p)
!DEC$      ATTRIBUTES C, ALIAS:'_Ptr_Sub' :: Ptr_Sub
           INTEGER p
         END SUBROUTINE Ptr_Sub
      END INTERFACE
      REAL A[10], VAR[10]
      POINTER (p, VAR)  ! VAR is the pointer-based
                        ! variable, p is the integer
                        ! pointer
      p = LOC(A)
      CALL Ptr_Sub (p)
      WRITE(*,*) "A(4) = ", A(4)
      END
! C subprogram
void Ptr_Sub (int *p)
{  float a[10];
  a[3] = 23.5;
  *p = a;
}
```

%LOC

Built-in Function: *Computes the internal address of a storage item.*

Syntax

```
result = %LOC (a)
```

a (Input) Is the name of an actual argument. It must be a variable, an expression, or the name of a procedure. It must not be the name of a statement function.

Description

The %LOC function produces an integer value that represents the location of the given argument. The value is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. You can use this integer value as an item in an arithmetic expression.

The LOC intrinsic function serves the same purpose as the %LOC built-in function.

LOG

Elemental Intrinsic Function (Generic):
Returns the natural logarithm of the argument.

Syntax

```
result = LOG (x)
```

x (Input) Must be of type real or complex. If *x* is real, its value must be greater than zero. If *x* is complex, its value must not be zero.

Results

The result type is the same as *x*. The result value is approximately equal to $\log_e x$.

If the arguments are complex, the result is the principal value with imaginary part omega in the range -pi <= omega <= pi.

If the real part of *x* < 0 and the imaginary part of *x* is a positive real zero, the imaginary part of the result is pi.

If the real part of *x* < 0 and the imaginary part of *x* is a negative real zero, the imaginary part of the result is -pi.

Specific Name	Argument Type	Result Type
ALOG [1,2]	REAL(4)	REAL(4)
DLOG	REAL(8)	REAL(8)
QLOG	REAL(16)	REAL(16)

Specific Name	Argument Type	Result Type
CLOG [2]	COMPLEX(4)	COMPLEX(4)
CDLOG[3,4]	COMPLEX(8)	COMPLEX(8)
CQLOG	COMPLEX(16)	COMPLEX(16)

[1]This function is treated like LOG.

[2]The setting of compiler options specifying real size can affect ALOG, LOG, and CLOG.

[3]This function can also be specified as ZLOG.

[4]The setting of compiler options specifying double size can affect CDLOG.

Example

LOG (8.0) has the value 2.079442.

LOG (25.0) has the value 3.218876.

The following shows another example:

```
REAL r

r = LOG(10.0)  ! returns 2.302585
```

See Also

- J to L
- EXP
- LOG10

LOG10

Elemental Intrinsic Function (Generic):
Returns the common logarithm of the argument.

Syntax

```
result = LOG10 (x)
```

x (Input) Must be of type real or complex. If *x* is real, its value must be greater than zero. If *x* is complex, its value must not be zero.

Results

The result type is the same as x. The result value is approximately equal to $\log_{10} x$.

Specific Name	Argument Type	Result Type
ALOG10 [1,2]	REAL(4)	REAL(4)
DLOG10 [3]	REAL(8)	REAL(8)
QLOG10	REAL(16)	REAL(16)
CLOG10 [2]	COMPLEX(4)	COMPLEX(4)
CDLOG10 [3]	COMPLEX(8)	COMPLEX(8)
CQLOG10	COMPLEX(16)	COMPLEX(16)

[1] This function is treated like LOG10.

[2] The setting of compiler options specifying real size can affect ALOG10, CLOG10, and LOG10.

[3] The setting of compiler options specifying double size can affect DLOG10 and CDLOG10.

Example

LOG10 (8.0) has the value 0.9030900.

LOG10 (15.0) has the value 1.176091.

The following shows another example:

```
REAL r
r = LOG10(10.0)  ! returns 1.0
```

See Also

- J to L
- LOG

LOGICAL Statement

Statement: *Specifies the LOGICAL data type.*

Syntax

```
LOGICAL
LOGICAL([KIND=] n)
LOGICAL*n
```

n Is kind 1, 2, 4, or 8.

If a kind parameter is specified, the logical constant has the kind specified. If no kind parameter is specified, the kind of the constant is default logical.

Example

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (2), SAVE :: doit, dont=.FALSE.
LOGICAL switch
! An equivalent declaration is:
LOGICAL flag1, flag2
LOGICAL (2) doit, dont=.FALSE.
ALLOCATABLE flag1, flag2
SAVE doit, dont
```

See Also

- J to L
- Logical Data Types
- Logical Constants
- Data Types, Constants, and Variables

LOGICAL Function

Elemental Intrinsic Function (Generic): *Converts the logical value of the argument to a logical value with different kind parameters.*

Syntax

```
result = LOGICAL (l[,kind])
```

l	(Input) Must be of type logical.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is of type logical. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter is that of default logical. The result value is that of *l*.

The setting of compiler options specifying integer size can affect this function.

Example

LOGICAL (L .OR. .NOT. L) has the value true and is of type default logical regardless of the kind parameter of logical variable L.

LOGICAL (.FALSE., 2) has the value false, with the kind parameter of INTEGER(KIND=2).

See Also

- J to L
- CMPLX
- INT
- REAL
- Logical Data Types

LONG

Portability Function: *Converts an INTEGER(2) argument to INTEGER(4) type.*

Module

USE IFPORT

Syntax

```
result = LONG (int2)
```

int2 (Input) INTEGER(2). Value to be converted.

Results

The result type is INTEGER(4). The result is the value of *int2* with type INTEGER(4). The upper 16 bits of the result are zeros and the lower 16 are equal to *int2*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- J to L
- INT
- KIND

Building Applications: Portability Library Overview

LOOP COUNT

General Compiler Directive: *Specifies the iterations (count) for a DO loop.*

Syntax

```
cDEC$ LOOP COUNT (n1[,n2]...)
cDEC$ LOOP COUNT= n1[,n2]...
cDEC$ LOOP COUNT MAX(n1), MIN(n1), AVG(n1)
cDEC$ LOOP COUNT MAX=n1, MIN=n1, AVG=n1
```

c Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

n1, *n2* Is a non-negative integer constant.

The value of the loop count affects heuristics used in software pipelining, vectorization, and loop-transformations.

Argument Form	Description
n1 [, *n2*]	Indicates that the next DO loop will iterate *n1*, *n2*, or some other number of times.
MAX, MIN, and AVG	Indicates that the next DO loop has the specified maximum, minimum, and average number (*n1*) of iterations.

Example

Consider the following:

```
cDEC$ LOOP COUNT (10000)
do i =1,m

b(i) = a(i) +1 ! This is likely to enable the loop to get software-pipelined

enddo
```

Note that you can specify more than one LOOP COUNT directive for a DO loop. For example, the following directives are valid:

```
!DEC$ LOOP COUNT (10, 20, 30)

!DEC$ LOOP COUNT MAX=100, MIN=3, AVG=17

DO

...
```

See Also

- J to L
- Rules for General Directives that Affect DO Loops

Optimizing Applications: Loop Count and Loop Distribution

LSHIFT

Elemental Intrinsic Function (Generic): *Shifts the bits in an integer left by a specified number of positions. See ISHFT.*

LSTAT

Portability Function: *Returns detailed information about a file.*

Module

USE IFPORT

Syntax

`result = LSTAT (name,statb)`

name	(Input) Character*(*). Name of the file to examine.
statb	(Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. See STATfor the possible values returned in *statb*.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code (see IERRNO).

LSTAT returns detailed information about the file named in *name*.

On Linux* and Mac OS* X systems, if the file denoted by *name* is a link, LSTAT provides information on the link, while STAT provides information on the file at the destination of the link.

On Windows* systems, LSTAT returns exactly the same information as STAT (because there are no symbolic links on these systems). STAT is the preferred function.

INQUIRE also provides information about file properties.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) info_array(12), istatus
character*20 file_name
print *, "Enter name of file to examime: "
read *, file_name
ISTATUS = LSTAT (file_name, info_array)
if (.NOT. ISTATUS) then
   print *, info_array
else
   print *, 'Error ',istatus
end if
```

See Also

- J to L
- INQUIRE
- GETFILEINFOQQ
- STAT
- FSTAT

Building Applications: Portability Library Overview

LTIME

Portability Subroutine: *Returns the components of the local time zone time in a nine-element array.*

Module

USE IFPORT

Syntax

CALL LTIME (*time*,*array*)

time	(Input) INTEGER(4). An elapsed time in seconds since 00:00:00 Greenwich mean time, January 1, 1970.

array (Output) INTEGER(4). One-dimensional array with 9 elements to contain local date and time data derived from *time*.
The elements of *array* are returned as follows:

Element	Value
array(1)	Seconds (0 - 59)
array(2)	Minutes (0 - 59)
array(3)	Hours (0 - 23)
array(4)	Day of month (1 - 31)
array(5)	Month (0 - 11)
array(6)	Years since 1900
array(7)	Day of week (0 - 6, where 0 is Sunday)
array(8)	Day of year (1 - 365)
array(9)	1 if daylight saving time is in effect; otherwise, 0.

CAUTION. This subroutine is not year-2000 compliant, use DATE_AND_TIME instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) input_time, time_array(9)
!   find number of seconds since 1/1/70
input_time=TIME()
!   convert number of seconds to time array
CALL LTIME (input_time, time_array)
PRINT *, time_array
```

See Also

- J to L
- DATE_AND_TIME

Building Applications: Portability Library Overview

M to N

MAKEDIRQQ

Portability Function: *Creates a new directory with a specified name.*

Module

USE IFPORT

Syntax

`result = MAKEDIRQQ (dirname)`

dirname (Input) Character*(*). Name of directory to be created.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

MAKEDIRQQ can create only one directory at a time. You cannot create a new directory and a subdirectory below it in a single command. MAKEDIRQQ does not translate path delimiters. You can use either slash (/) or backslash (\) as valid delimiters.

If an error occurs, call GETLASTERRORQQ to retrieve the error message. Possible errors include:

- ERR$ACCES - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- ERR$EXIST - The directory already exists.
- ERR$NOENT - The file or path specified was not found.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

LOGICAL(4) result

result = MAKEDIRQQ('mynewdir')

IF (result) THEN

   WRITE (*,*) 'New subdirectory successfully created'

ELSE

   WRITE (*,*) 'Failed to create subdirectory'

END IF

END
```

See Also

- M to N
- DELDIRQQ
- CHANGEDIRQQ
- GETLASTERRORQQ

MALLOC

Elemental Intrinsic Function (Generic):
Allocates a block of memory. This is a generic function that has no specific function associated with it. It must not be passed as an actual argument.

Syntax

```
result = MALLOC (size)
```

size (Input) Must be of type integer. This value is the size (in bytes) of memory to be allocated.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The result is the starting address of the allocated memory. The memory allocated can be freed by using the FREE intrinsic function.

Example

```
INTEGER(4) SIZE
REAL(4) STORAGE(*)
POINTER (ADDR, STORAGE)     ! ADDR will point to STORAGE
SIZE = 1024                 ! Size in bytes
ADDR = MALLOC(SIZE)         ! Allocate the memory
CALL FREE(ADDR)             ! Free it
```

MAP...END MAP

Statement: *Specifies mapped field declarations that are part of a UNION declaration within a STRUCTURE declaration. See STRUCTURE.*

Example

```
UNION
  MAP
     CHARACTER*20 string
  END MAP
  MAP
     INTEGER*2 number(10)
  END MAP
END UNION
UNION
  MAP
     RECORD /Cartesian/ xcoord, ycoord
  END MAP
  MAP
     RECORD /Polar/ length, angle
  END MAP
END UNION
```

MASTER

OpenMP* Fortran Compiler Directive: *Specifies a block of code to be executed by the master thread of the team.*

Syntax

```
c$OMP MASTER
   block
```

```
c$OMP END MASTER
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

When the MASTER directive is specified, the other threads in the team skip the enclosed block (section) of code and continue execution. There is no implied barrier, either on entry to or exit from the master section.

Example

The following example forces the master thread to execute the routines OUTPUT and INPUT:

```
c$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
c$OMP MASTER
      CALL OUTPUT(X)
      CALL INPUT(Y)
c$OMP END MASTER
      CALL WORK(Y)
c$OMP END PARALLEL
```

See Also

- M to N
- OpenMP Fortran Compiler Directives

MATMUL

Transformational Intrinsic Function (Generic):
Performs matrix multiplication of numeric or logical matrices.

Syntax

```
result = MATMUL (matrix_a,matrix_b)
```

matrix_a	(Input) Must be an array of rank one or two. It must be of numeric (integer, real, or complex) or logical type.

matrix_b (Input) Must be an array of rank one or two. It must be of numeric type if *matrix_a* is of numeric type or logical type if *matrix_a* is logical type.
At least one argument must be of rank two. The size of the first (or only) dimension of *matrix_b* must equal the size of the last (or only) dimension of *matrix_a*.

Results

The result is an array whose type depends on the data type of the arguments, according to the rules described in Data Type of Numeric Expressions. The rank and shape of the result depends on the rank and shapes of the arguments, as follows:

- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m, k), the result is a rank-two array with shape (n, k).
- If *matrix_a* has shape (m) and *matrix_b* has shape (m, k), the result is a rank-one array with shape (k).
- If *matrix_a* has shape (n, m) and *matrix_b* has shape (m), the result is a rank-one array with shape (n).

If the arguments are of numeric type, element (i, j) of the result has the value SUM((row i of *matrix_a*) * (column j of *matrix_b*)). If the arguments are of logical type, element (i, j) of the result has the value ANY((row i of *matrix_a*) .AND. (column j of *matrix_b*)).

Example

A is matrix

```
[ 2  3  4 ]
[ 3  4  5 ],
```

B is matrix

```
[ 2  3 ]
[ 3  4 ]
[ 4  5 ],
```

X is vector (1, 2), and Y is vector (1, 2, 3).

The result of MATMUL (A, B) is the matrix-matrix product AB with the value

```
[ 29  38 ]
[ 38  50 ].
```

The result of MATMUL (X, A) is the vector-matrix product XA with the value (8, 11, 14).

The result of MATMUL (A, Y) is the matrix-vector product AY with the value (20, 26).

The following shows another example:

```
INTEGER a(2,3), b(3,2), c(2), d(3), e(2,2), f(3), g(2)
a = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  a is   1 3 5
!         2 4 6
b = RESHAPE((/1, 2, 3, 4, 5, 6/), (/3, 2/))
!  b is   1 4
!         2 5
!         3 6
c = (/1, 2/)      ! c is [1 2]
d = (/1, 2, 3/)   ! d is [1 2 3]
e = MATMUL(a, b)    ! returns 22 49
                    !         28 64
f = MATMUL(c,a)   ! returns [5 11 17]
g = MATMUL(a,d)   ! returns [22 28]
WRITE(*,*) e, f, g
END
```

See Also

- M to N
- TRANSPOSE
- PRODUCT

MAX

Elemental Intrinsic Function (Generic):
Returns the maximum value of the arguments.

Syntax

```
result = MAX (a1,a2[,a3]...)
```

a1, a2, a3 (Input) All must have the same type (integer or real) and kind parameters.

Results

For MAX0, AMAX1, DMAX1, QMAX1, IMAX0, JMAX0, and KMAX0, the result type is the same as the arguments. For MAX1, IMAX1, JMAX1, and KMAX1, the result type is integer. For AMAX0, AIMAX0, AJMAX0, and AKMAX0, the result type is real. The value of the result is that of the largest argument.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IMAX0	INTEGER(2)	INTEGER(2)
AIMAX0	INTEGER(2)	REAL(4)
MAX0 [2]	INTEGER(4)	INTEGER(4)
AMAX0 [3, 4]	INTEGER(4)	REAL(4)
KMAX0	INTEGER(8)	INTEGER(8)
AKMAX0	INTEGER(8)	REAL(4)
IMAX1	REAL(4)	INTEGER(2)
MAX1 [4, 5, 6]	REAL(4)	INTEGER(4)
KMAX1	REAL(4)	INTEGER(8)
AMAX1 [7]	REAL(4)	REAL(4)
DMAX1	REAL(8)	REAL(8)
QMAX1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMAX0.

[3]Or AJMAX0.AMAX0 is the same as REAL (MAX).

Specific Name [1]	Argument Type	Result Type

[4]In Fortran 95/90, AMAX0 and MAX1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[5]Or JMAX1.MAX1 is the same as INT(MAX).

[6]The setting of compiler options specifying integer size can affect MAX1.

[7]The setting of compiler options specifying real size can affect AMAX1.

Example

MAX (2.0, -8.0, 6.0) has the value 6.0.

MAX (14, 32, -50) has the value 32.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MAX(5, 6, 7)            ! returns 7
m2 = MAX1(5.7, 3.2, -8.3)    ! returns 5
r1 = AMAX0(5, 6, 7)          ! returns 7.0
r2 = AMAX1(6.4, -12.2, 4.9) ! returns 6.4
```

See Also

- M to N
- MIN

MAXEXPONENT

Inquiry Intrinsic Function (Generic): *Returns the maximum exponent in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = MAXEXPONENT (x)
```

x (Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value e_{max}, as defined in Model for Real Data.

Example

```
REAL(4) x
INTEGER i
i = MAXEXPONENT(x)     ! returns 128.
```

See Also

- M to N
- MINEXPONENT

MAXLOC

Transformational Intrinsic Function (Generic):
Returns the location of the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

```
result = MAXLOC (array [, dim] [, mask] [, kind])
```

array	(Input) Must be an array of type integer or real.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. This argument is a Fortran 95 feature.
mask	(Input; optional) Must be a logical array that is conformable with *array*.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array*.
- If MAXLOC(*array*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value in *array*.

  The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*.
- If MAXLOC(*array*, MASK= *mask*) is specified, the elements in the array result form the subscript of the location of the element with the maximum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2,...d_{dim-1}, d_{dim+1},...d_n), where (d_1, d_2,...d_n) is the shape of *array*.
- If *array* has rank one, MAXLOC(*array*, *dim*[, *mask*]) has a value equal to that of MAXLOC(*array*[, MASK = *mask*]). Otherwise, the value of element (s_1, s_2,...s_{dim-1}, s_{dim+1},...s_n) of MAXLOC(*array*, *dim*[, *mask*]) is equal to MAXLOC(*array*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n) [, MASK = *mask*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n)]).

If more than one element has maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is controlled by compiler option `assume [no]old_maxminloc`, which can set the result to either 1 or 0.

The setting of compiler options specifying integer size can affect this function.

Example

The value of MAXLOC((/3, 7, 4, 7/)) is (2), which is the subscript of the location of the first occurrence of the maximum value in the rank-one array.

A is the array

```
[  4    0   -3    2 ]
[  3    1   -2    6 ]
[ -1   -4    5   -5 ].
```

MAXLOC (A, MASK=A .LT. 5) has the value (1, 1) because these are the subscripts of the location of the maximum value (4) that is less than 5.

MAXLOC (A, DIM=1) has the value (1, 2, 3, 2). 1 is the subscript of the location of the maximum value (4) in column 1; 2 is the subscript of the location of the maximum value (1) in column 2; and so forth.

MAXLOC (A, DIM=2) has the value (1, 4, 3). 1 is the subscript of the location of the maximum value in row 1; 4 is the subscript of the location of the maximum value in row 2; and so forth.

The following shows another example:

```
 INTEGER i, max
 INTEGER i, maxl(1)
 INTEGER array(3, 3)
 INTEGER, ALLOCATABLE :: AR1(:)
 ! put values in array
 array = RESHAPE((/7, 9, -1, -2, 5, 0, 3, 6, 9/),     &
                 (/3, 3/))
 ! array is  7 -2 3
 !           9 5 6
 !          -1 0 9
 i = SIZE(SHAPE(array))   ! Get number of dimensions
                          ! in array
 ALLOCATE ( AR1(i))       ! Allocate AR1 to number
                          ! of dimensions in array
 AR1 = MAXLOC (array, MASK = array .LT. 7) ! Get
                          ! the location (subscripts) of
                          ! largest element less than 7
                          ! in array
 !
 ! MASK = array .LT. 7 creates a mask array the same
 ! size and shape as array whose elements are .TRUE. if
 ! the corresponding element in array is less than 7,
 ! and .FALSE. if it is not. This mask causes MAXLOC to
 ! return the index of the element in array with the
 ! greatest value less than 7.
 !
 ! array is  7 -2 3 and MASK=array .LT. 7 is  F T T
 !           9  5 6                           F T T
```

```
!          -1  0 9                          T T F
! and AR1 = MAXLOC(array, MASK = array .LT. 7) returns
! (2, 3), the location of the element with value 6
maxl = MAXLOC((/1, 4, 3, 4/))    ! returns 2, the first
                                 ! occurrence of maximum
END
```

See Also

- M to N
- MAXVAL
- MINLOC
- MINVAL

MAXVAL

Transformational Intrinsic Function (Generic):
Returns the maximum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MAXVAL (array[,dim] [,mask])`

array	(Input) Must be an array of type integer or real.
dim	(Input; optional) Must be a scalar integer expression with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MAXVAL(*array*) is specified, the result has a value equal to the maximum value of all the elements in *array*.

- If MAXVAL(*array*, MASK= *mask*) is specified, the result has a value equal to the maximum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- An array result has a rank that is one less than *array*, and shape (d_1, d_2,...,d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- If *array* has rank one, MAXVAL(*array*, *dim*[, *mask*]) has a value equal to that of MAXVAL(*array*[,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of MAXVAL(*array*, *dim*, [, *mask*]) is equal to MAXVAL(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array*.

Example

The value of MAXVAL ((/2, 3, 4/)) is 4 because that is the maximum value in the rank-one array.

MAXVAL (B, MASK=B .LT. 0.0) finds the maximum value of the negative elements of B.

C is the array

```
[ 2  3  4 ]
[ 5  6  7 ].
```

MAXVAL (C, DIM=1) has the value (5, 6, 7). 5 is the maximum value in column 1; 6 is the maximum value in column 2; and so forth.

MAXVAL (C, DIM=2) has the value (4, 7). 4 is the maximum value in row 1 and 7 is the maximum value in row 2.

The following shows another example:

```
INTEGER array(2,3), i(2), max
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/),(/2, 3/))
! array is   1 5 3
!            4 2 6
i = SHAPE(array)      ! i = [2 3]
ALLOCATE (AR1(i(2)))  ! dimension AR1 to the number of
                      ! elements in dimension 2
                      ! (a column) of array
ALLOCATE (AR2(i(1)))  ! dimension AR2 to the number of
                      ! elements in dimension 1
                      ! (a row) of array
max = MAXVAL(array, MASK = array .LT. 4) ! returns 3
AR1 = MAXVAL(array, DIM = 1)  ! returns [ 4 5 6 ]
AR2 = MAXVAL(array, DIM = 2)  ! returns [ 5 6 ]
END
```

See Also

- M to N
- MAXLOC
- MINVAL
- MINLOC

MBCharLen (W*32, W*64)

NLS Function: *Returns the length, in bytes, of the first character in a multibyte-character string.*

Module

USE IFNLS

Syntax

```
result = MBCharLen (string)
```

string	(Input) Character*(*). String containing the character whose length is to be determined. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of bytes in the first character contained in *string*. The function returns 0 if *string* has no characters (is length 0).

MBCharLen does not test for multibyte character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBCurMax
- MBLead
- MBLen
- MBLen_Trim

MBConvertMBToUnicode (W*32, W*64)

NLS Function: *Converts a multibyte-character string from the current codepage to a Unicode string.*

Module

USE IFNLS

Syntax

```
result = MBConvertMBToUnicode (mbstr,unicodestr[,flags])
```

mbstr	(Input) Character*(*). Multibyte codepage string to be converted.
unicodestr	(Output) INTEGER(2). Array of integers that is the translation of the input string into Unicode.

flags (Input; optional) INTEGER(4). If specified, modifies the string conversion. If *flags* is omitted, the value NLS$Precomposed is used. Available values (defined in IFNLS.F90) are:

- NLS$Precomposed - Use precomposed characters always. (default)
- NLS$Composite - Use composite wide characters always.
- NLS$UseGlyphChars - Use glyph characters instead of control characters.
- NLS$ErrorOnInvalidChars - Returns -1 if an invalid input character is encountered.

The flags NLS$Precomposed and NLS$Composite are mutually exclusive. You can combine NLS$UseGlyphChars with either NLS$Precomposed or NLS$Composite using an inclusive OR (IOR or OR).

Results

The result type is INTEGER(4). If no error occurs, the result is the number of bytes written to *unicodestr*(bytes are counted, not characters), or the number of bytes required to hold the output string if *unicodestr* has zero size. If the *unicodestr* array is bigger than needed to hold the translation, the extra elements are set to space characters. If *unicodestr* has zero size, the function returns the number of bytes required to hold the translation and nothing is written to *unicodestr*.

If an error occurs, one of the following negative values is returned:

- NLS$ErrorInsufficentBuffer - The *unicodestr* argument is too small, but not zero size so that the needed number of bytes would be returned.
- NLS$ErrorInvalidFlags - The *flags* argument has an illegal value.
- NLS$ErrorInvalidCharacter - A character with no Unicode translation was encountered in *mbstr*. This error can occur only if the NLS$InvalidCharsError flag was used in *flags*.
-

NOTE. By default, or if *flags* is set to NLS$Precomposed, the function MBConvertMBToUnicode attempts to translate the multibyte codepage string to a precomposed Unicode string. If a precomposed form does not exist, the function attempts to translate the codepage string to a composite form.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBConvertUnicodeToMB

MBConvertUnicodeToMB (W*32, W*64)

NLS Function: *Converts a Unicode string to a multibyte-character string from the current codepage.*

Module

USE IFNLS

Syntax

`result = MBConvertUnicodeToMB (unicodestr,mbstr[,flags])`

unicodestr	(Input) INTEGER(2). Array of integers holding the Unicode string to be translated.
mbstr	(Output) Character*(*). Translation of Unicode string into multibyte character string from the current codepage.
flags	(Input; optional) INTEGER(4). If specified, argument to modify the string conversion. If *flags* is omitted, no extra checking of the conversion takes place. Available values (defined in IFNLS.F90) are: • NLS$CompositeCheck - Convert composite characters to precomposed. • NLS$SepChars - Generate separate characters. • NLS$DiscardDns - Discard nonspacing characters. • NLS$DefaultChars - Replace exceptions with default character. The last three flags (NLS$SepChars, NLS$DiscardDns, and NLS$DefaultChars) are mutually exclusive and can be used only if NLS$CompositeCheck is set, in which case one (and only one) of them is combined with NLS$CompositeCheck using an inclusive

OR (IOR or OR). These flags determine what translation to make when there is no precomposed mapping for a base character/nonspace character combination in the Unicode wide character string. The default (IOR(NLS$CompositeCheck, NLS$SepChars)) is to generate separate characters.

Results

The result type is INTEGER(4). If no error occurs, returns the number of bytes written to *mbstr*(bytes are counted, not characters), or the number of bytes required to hold the output string if *mbstr* has zero length. If *mbstr* is longer than the translation, it is blank-padded. If *mbstr* is zero length, the function returns the number of bytes required to hold the translation and nothing is written to *mbstr*.

If an error occurs, one of the following negative values is returned:

- NLS$ErrorInsufficentBuffer - The *mbstr* argument is too small, but not zero length so that the needed number of bytes is returned.
- NLS$ErrorInvalidFlags - The *flags* argument has an illegal value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBConvertMBToUnicode

MBCurMax (W*32, W*64)

NLS Function: *Returns the longest possible multibyte character length, in bytes, for the current codepage.*

Module

USE IFNLS

Syntax

```
result = MBCurMax( )
```

Results

The result type is INTEGER(4). The result is the longest possible multibyte character, in bytes, for the current codepage.

The MBLenMax parameter, defined in the module `IFNLS.F90`, is the longest length, in bytes, of any character in any codepage installed on the system.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBCharLen

Building Applications: MBCS Inquiry Routines

MBINCHARQQ (W*32, W*64)

NLS Function: *Performs the same function as INCHARQQ except that it can read a single multibyte character at once, and it returns the number of bytes read as well as the character.*

Module

USE IFNLS

Syntax

`result = MBINCHARQQ (`*`string`*`)`

`string`	(Output) CHARACTER(MBLenMax). String containing the read characters, padded with blanks up to the length MBLenMax. The MBLenMax parameter, defined in the module `IFNLS.F90`, is the longest length, in bytes, of any character in any codepage installed on the system.

Results

The result type is INTEGER(4). The result is the number of characters read.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- INCHARQQ
- MBCurMax
- MBCharLen
- MBLead

MBINDEX (W*32, W*64)

NLS Function: *Performs the same function as*INDEX *except that the strings manipulated can contain multibyte characters.*

Module

USE IFNLS

Syntax

```
result = MBINDEX (string,substring[,back])
```

string	(Input) CHARACTER*(*). String to be searched for the presence of *substring*. Can contain multibyte characters.
substring	(Input) CHARACTER*(*). Substring whose position within *string* is to be determined. Can contain multibyte characters.
back	(Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results

The result type is INTEGER(4). If *back* is omitted or is .FALSE., it returns the leftmost position in *string* that contains the start of *substring*. If *back* is .TRUE., it returns the rightmost position in *string* that contains the start of *substring*. If *string* does not contain *substring*, it returns 0. If *substring* occurs more than once, it returns the starting position of the first occurrence ("first" is determined by the presence and value of *back*).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- INDEX
- MBSCAN
- MBVERIFY

MBJISToJMS, MBJMSToJIS (W*32, W*64)

NLS Functions: *Converts Japan Industry Standard (JIS) characters to Microsoft Kanji (JMS) characters, or converts JMS characters to JIS characters.*

Module

USE IFNLS

Syntax

```
result = MBJISToJMS (char)
```

```
result = MBJISToJIS (char)
```

char	(Input) CHARACTER(2). JIS or JMS character to be converted. A JIS character is converted only if the lead and trail bytes are in the hexadecimal range 21 through 7E. A JMS character is converted only if the lead byte is in the hexadecimal range 81 through 9F or E0 through FC, and the trail byte is in the hexadecimal range 40 through 7E or 80 through FC.

Results

The result type is character with length 2. MBJISToJMS returns a Microsoft Kanji (Shift JIS or JMS) character. MBJMSToJIS returns a Japan Industry Standard (JIS) character.

Only computers with Japanese installed as one of the available languages can use the MBJISToJMS and MBJMSToJIS conversion functions.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N

- NLSEnumLocales
- NLSEnumCodepages
- NLSGetLocale
- NLSSetLocale

Building Applications: MBCS Conversion Routines

MBLead (W*32, W*64)

NLS Function: *Determines whether a given character is the lead (first) byte of a multibyte character sequence.*

Module

USE IFNLS

Syntax

`result = MBLead (char)`

char (Input) CHARACTER(1). Character to be tested for lead status.

Results

The result type is LOGICAL(4). The result is .TRUE. if *char* is the first character of a multibyte character sequence; otherwise, .FALSE..

MBLead only works stepping forward through a whole multibyte character string. For example:

```
      DO i = 1, LEN(str)  ! LEN returns the number of bytes, not the
                          !  number of characters in str
         WRITE(*, 100) MBLead (str(i:i))
      END DO
100   FORMAT (L2, \)
```

MBLead is passed only one character at a time and must start on a lead byte and step through a string to establish context for the character. MBLead does not correctly identify a nonlead byte if it is passed only the second byte of a multibyte character because the status of lead byte or trail byte depends on context.

The function MBStrLead is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context. So, MBStrLead can be much slower than MBLead (up to *n* times slower, where *n* is the length of the string).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBStrLead
- MBCharLen

Building Applications: MBCS Inquiry Routines

MBLen (W*32, W*64)

NLS Function: *Returns the number of characters in a multibyte-character string, including trailing blanks.*

Module

USE IFNLS

Syntax

```
result = MBLen (string)
```

string	(Input) CHARACTER*(*). String whose characters are to be counted. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of characters in *string*.

MBLen recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBLen_Trim
- MBStrLead

Building Applications: MBCS Inquiry Routines

MBLen_Trim (W*32, W*64)

NLS Function: *Returns the number of characters in a multibyte-character string, not including trailing blanks.*

Module

USE IFNLS

Syntax

```
result = MBLen_Trim (string)
```

string	(Input) Character*(*). String whose characters are to be counted. Can contain multibyte characters.

Results

The result type is INTEGER(4). The result is the number of characters in *string* minus any trailing blanks (blanks are bytes containing character 32 (hex 20) in the ASCII collating sequence).

MBLen_Trim recognizes multibyte-character sequences according to the multibyte codepage currently in use. It does not test for multibyte-character validity.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBLen
- MBStrLead

Building Applications: MBCS Inquiry Routines

MBLGE, MBLGT, MBLLE, MBLLT, MBLEQ, MBLNE (W*32, W*64)

NLS Functions: *Perform the same functions as LGE, LGT, LLE, LLT and the logical operators .EQ. and .NE. except that the strings being compared can include multibyte characters, and optional flags can modify the comparison.*

Module

USE IFNLS

Syntax

```
result = MBLGE (string_a,string_b, [ flags])
result = MBLGT (string_a,string_b, [ flags])
result = MBLLE (string_a,string_b, [ flags])
result = MBLLT (string_a,string_b, [ flags])
result = MBLEQ (string_a,string_b, [ flags])
result = MBLNE (string_a,string_b, [ flags])
```

string_a, *string_b* — (Input) Character*(*). Strings to be compared. Can contain multibyte characters.

flags — (Input; optional) INTEGER(4). If specified, determines which character traits to use or ignore when comparing strings. You can combine several flags using an inclusive OR (IOR or OR). There are no illegal combinations of flags, and the functions may be used without flags, in which case all flag options are turned off. The available values (defined in `IFNLS.F90`) are:

- NLS$MB_IgnoreCase - Ignore case.
- NLS$MB_IgnoreNonspace - Ignore nonspacing characters (this flag removes Japanese accent characters if they exist).
- NLS$MB_IgnoreSymbols - Ignore symbols.
- NLS$MB_IgnoreKanaType - Do not differentiate between Japanese Hiragana and Katakana characters (corresponding Hiragana and Katakana characters will compare as equal).

- NLS$MB_IgnoreWidth - Do not differentiate between a single-byte character and the same character as a double byte.
- NLS$MB_StringSort - Sort all symbols at the beginning, including the apostrophe and hyphen (see the Notebelow).

Results

The result type is LOGICAL(4). Comparisons are made using the current locale, not the current codepage. The codepage used is the default for the language/country combination of the current locale.

The results of these functions are as follows:

- MBLGE returns .TRUE. if the strings are equal or *string_a* comes last in the collating sequence; otherwise, .FALSE..
- MBLGT returns .TRUE. if *string_a* comes last in the collating sequence; otherwise, .FALSE..
- MBLLE returns .TRUE. if the strings are equal or *string_a* comes first in the collating sequence; otherwise, .FALSE..
- MBLLT returns .TRUE. if *string_a* comes first in the collating sequence; otherwise, .FALSE..
- MBLEQ returns .TRUE. if the strings are equal in the collating sequence; otherwise, .FALSE..
- MBLNE returns .TRUE. if the strings are not equal in the collating sequence; otherwise, .FALSE..

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, then the return value indicates that the longer string is greater.

If *flags* is invalid, the functions return .FALSE..

If the strings supplied contain Arabic Kashidas, the Kashidas are ignored during the comparison. Therefore, if the two strings are identical except for Kashidas within the strings, the functions return a value indicating they are "equal" in the collation sense, though not necessarily identical.

NOTE. When not using the NLS$MB_StringSort flag, the hyphen and apostrophe are special symbols and are treated differently than others. This is to ensure that words like coop and co-op stay together within a list. All symbols, except the hyphen and apostrophe, sort before any other alphanumeric character. If you specify the NLS$MB_StringSort flag, hyphen and apostrophe sort at the beginning also.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- LGE
- LGT
- LLE
- LLT

MBNext (W*32, W*64)

NLS Function: *Returns the position of the first lead byte or single-byte character immediately following the given position in a multibyte-character string.*

Module

USE IFNLS

Syntax

`result = MBNext (string,position)`

string	(Input) Character*(*). String to be searched for the first lead byte or single-byte character after the current position. Can contain multibyte characters.
position	(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or a single-byte character. Cannot be the position of a trail (second) byte of a multibyte character.

Results

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately following the position given in *position*, or 0 if no following first byte is found in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBPrev

Building Applications: MBCS Inquiry Routines

MBPrev (W*32, W*64)

NLS Function: *Returns the position of the first lead byte or single-byte character immediately preceding the given string position in a multibyte-character string.*

Module

USE IFNLS

Syntax

`result = MBPrev (string,position)`

string	(Input) Character*(*). String to be searched for the first lead byte or single-byte character before the current position. Can contain multibyte characters.
position	(Input) INTEGER(4). Position in *string* to search from. Must be the position of a lead byte or single-byte character. Cannot be the position of the trail (second) byte of a multibyte character.

Results

The result type is INTEGER(4). The result is the position of the first lead byte or single-byte character in *string* immediately preceding the position given in *position*, or 0 if no preceding first byte is found in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBNext

Building Applications: MBCS Inquiry Routines

MBSCAN (W*32, W*64)

NLS Function: *Performs the same function as*SCAN *except that the strings manipulated can contain multibyte characters.*

Module

USE IFNLS

Syntax

`result = MBSCAN (string,set[,back])`

string	(Input) Character*(*). String to be searched for the presence of any character in *set*.
set	(Input) Character*(*). Characters to search for.
back	(Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results

The result type is INTEGER(4). If *back* is .FALSE. or is omitted, it returns the position of the leftmost character in *string* that is in *set*. If *back* is .TRUE., it returns the rightmost character in *string* that is in *set*. If no characters in *string* are in *set*, it returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- SCAN
- MBINDEX
- MBVERIFY

MBStrLead (W*32, W*64)

NLS Function: *Performs a context-sensitive test to determine whether a given character byte in a string is a multibyte-character lead byte.*

Module

USE IFNLS

Syntax

`result = MBStrLead (string,position)`

string	(Input) Character*(*). String containing the character byte to be tested for lead status.
position	(Input) INTEGER(4). Position in *string* of the character byte in the string to be tested.

Results

The result type is LOGICAL(4). The result is .TRUE. if the character byte in *position* of *string* is a lead byte; otherwise, .FALSE..

MBStrLead is passed a whole string and can identify any byte within the string as a lead or trail byte because it performs a context-sensitive test, scanning all the way back to the beginning of a string if necessary to establish context.

MBLead is passed only one character at a time and must start on a lead byte and step through a string one character at a time to establish context for the character. So, MBStrLead can be much slower than MBLead (up to *n* times slower, where *n* is the length of the string).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- MBLead

Building Applications: MBCS Inquiry Routines

MBVERIFY (W*32, W*64)

NLS Function: *Performs the same function as*VERIFY *except that the strings manipulated can contain multibyte characters.*

Module

USE IFNLS

Syntax

`result = MBVERIFY (string,set[,back])`

string	(Input) Character*(*). String to be searched for presence of any character not in *set*.
set	(Input) Character*(*). Set of characters tested to verify that it includes all the characters in *string*.
back	(Input; optional) LOGICAL(4). If specified, determines direction of the search. If *back* is .FALSE. or is omitted, the search starts at the beginning of *string* and moves toward the end. If *back* is .TRUE., the search starts end of *string* and moves toward the beginning.

Results

The result type is INTEGER(4). If *back* is .FALSE. or is omitted, it returns the position of the leftmost character in *string* that is not in *set*. If *back* is .TRUE., it returns the rightmost character in *string* that is not in *set*. If all the characters in *string* are in *set*, it returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- VERIFY
- MBINDEX
- MBSCAN

MCLOCK

Inquiry Intrinsic Function (Specific): *Returns time accounting for a program.*

Syntax

```
result = MCLOCK( )
```

Results

The result type is INTEGER(4). The result is the sum (in units of microseconds) of the current process's user time and the user and system time of all its child processes.

MEMORYTOUCH (i64 only)

General Compiler Directive: *Ensures that a specific memory location is updated dynamically.*

Syntax

*c*DEC$ MEMORYTOUCH (*array*[, *schedule-type* [(*chunk-size*)]] [,*init-type*])

c	Is a c, C, !, or *. (See Syntax Rules for Compiler Directives.)
array	Is an array of type INTEGER(4), INTEGER(8), REAL(4) or REAL(8).
schedule-type	Is STATIC, GUIDED, RUNTIME or DYNAMIC, whichever is consistent with the OpenMP conforming processing of the subsequent parallel loops.
chunk-size	Is an integer expression.
init-type	Is LOAD or STORE. If *init-type* is LOAD, the compiler generates an OpenMP loop which fetches elements of *array* into a temporary variable. If *init-type* is STORE, the compiler generates an OpenMP loop which sets elements of *array* to zero.

The MEMORYTOUCH directive ensures that a specific memory location is updated dynamically. This prevents the possibility of multiple, simultaneous writing threads.

This directive supports correctly distributed memory initialization and NUMA pre-fetching.

To use this directive, OpenMP must be enabled.

Example

```
c$DEC memorytouch (ARRAY_A)c$DEC memorytouch (ARRAY_A, LOAD)c$DEC memorytouch (ARRAY_A,
STATIC (load+jf(3)) )c$DEC memorytouch (ARRAY_A, GUIDED (20), STORE)
```

See Also

- M to N
- General Compiler Directives

MEMREF_CONTROL (i64 only)

General Compiler Directive: *Lets you provide cache hints on prefetches, loads, and stores.*

Syntax

*c*DEC$ MEMREF_CONTROL *address1*[: *locality*[: *latency*]] [,*address2*...]

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
address1, *address2*	Is a memory reference. You must specify at least one *address*.
locality	Is an optional scalar integer initialization expression with the value 1, 2, 3, or 4. To use this argument, you must also specify *address*. This argument specifies the cache level at which this data has temporal locality, that is, where the data should be kept for future accesses. This will determine the load/store hint (or prefetch hint) to be used for this reference. The value can be one of the following: FOR_K_LOCALITY_L1 = 1 FOR_K_LOCALITY_L2 = 2 FOR_K_LOCALITY_L3 = 3 FOR_K_LOCALITY_MEM = 4
latency	Is an optional scalar integer initialization expression with the value 1, 2, 3, or 4. To use this argument, you must also specify *address* and *locality*.

This argument specifies the most appropriate latency value to be used for a load (or the latency that has to be overlapped if a prefetch is issued for this address). The value can be one of the following:

FOR_K_LATENCY_L1 = 1
FOR_K_LATENCY_L2 = 2
FOR_K_LATENCY_L3 = 3
FOR_K_LATENCY_MEM = 4

See Also

- M to N
- PREFETCH and NOPREFETCH

MERGE

Elemental Intrinsic Function (Generic): *Selects between two values or between corresponding elements in two arrays, according to the condition specified by a logical mask.*

Syntax

`result = MERGE (tsource,fsource,mask)`

tsource	(Input) May be of any data type.
fsource	(Input) Must be of the same type and type parameters as *tsource*.
mask	(Input) Must be of type logical.

Results

The result type is the same as *tsource*. The value of *mask* determines whether the result value is taken from *tsource* (if *mask* is true) or *fsource* (if *mask* is false).

Example

For MERGE (1.0, 0.0, R < 0), R = -3 has the value 1.0, and R = 7 has the value 0.0.

TSOURCE is the array

```
[ 1  3  5 ]
[ 2  4  6 ],
```

FSOURCE is the array

```
[ 8  9  0 ]
[ 1  2  3 ],
```

and MASK is the array

```
[ F  T  T]
[ T  T  F].
```

MERGE (TSOURCE, FSOURCE, MASK) produces the result:

```
[ 8  3  5 ]
[ 2  4  3 ].
```

The following shows another example:

```
INTEGER tsource(2, 3), fsource(2, 3), AR1 (2, 3)
LOGICAL mask(2, 3)
tsource = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2, 3/))
fsource = RESHAPE((/7, 0, 8, -1, 9, -2/), (/2, 3/))
mask = RESHAPE((/.TRUE., .FALSE., .FALSE., .TRUE.,         &
                .TRUE., .FALSE./), (/2,3/))
! tsource is  1 2 3 , fsource is  7  8  9 , mask is  T F T
!             4 5 6               0 -1 -2            F T F
AR1 = MERGE(tsource, fsource, mask) ! returns  1 8  3
                                    !          0 5 -2
END
```

MESSAGE

General Compiler Directive: *Specifies a character string to be sent to the standard output device during the first compiler pass; this aids debugging.*

Syntax

```
cDEC$ MESSAGE:string
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
string	Is a character constant specifying a message.

Example

```
!DEC$ MESSAGE:'Compiling Sound Speed Equations'
```

See Also

- M to N
- General Compiler Directives

MESSAGEBOXQQ (W*32, W*64)

QuickWin Function: *Displays a message box in a QuickWin window.*

Module

USE IFQWIN

Syntax

```
result = MESSAGEBOXQQ (msg,caption,mtype)
```

msg	(Input) Character*(*). Null-terminated C string. Message the box displays.
caption	(Input) Character*(*). Null-terminated C string. Caption that appears in the title bar.

mtype (Input) INTEGER(4). Symbolic constant that determines the objects (buttons and icons) and properties of the message box. You can combine several constants (defined in `IFQWIN.F90`) using an inclusive OR (IOR or OR). The symbolic constants and their associated objects or properties are as follows:

- MB$ABORTRETRYIGNORE - The Abort, Retry, and Ignore buttons.
- MB$DEFBUTTON1 - The first button is the default.
- MB$DEFBUTTON2 - The second button is the default.
- MB$DEFBUTTON3 - The third button is the default.
- MB$ICONASTERISK, MB$ICONINFORMATION - Lowercase *i* in blue circle icon.
- MB$ICONEXCLAMATION - The exclamation-mark icon.
- MB$ICONHAND, MB$ICONSTOP - The stop-sign icon.
- MB$ICONQUESTION - The question-mark icon.
- MB$OK - The OK button.
- MB$OKCANCEL - The OK and Cancel buttons.
- MB$RETRYCANCEL - The Retry and Cancel buttons.
- MB$SYSTEMMODAL - Box is system-modal: all applications are suspended until the user responds.
- MB$YESNO - The Yes and No buttons.
- MB$YESNOCANCEL - The Yes, No, and Cancel buttons.

Results

The result type is INTEGER(4). The result is zero if memory is not sufficient for displaying the message box. Otherwise, the result is one of the following values, indicating the user's response to the message box:

- MB$IDABORT - The Abort button was pressed.
- MB$IDCANCEL - The Cancel button was pressed.
- MB$IDIGNORE - The Ignore button was pressed.
- MB$IDNO - The No button was pressed.

- MB$IDOK - The OK button was pressed.
- MB$IDRETRY - The Retry button was pressed.
- MB$IDYES - The Yes button was pressed.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin app
USE IFQWIN
message = MESSAGEBOXQQ('Do you want to continue?'C,    &
          'Matrix'C,   &
          MB$ICONQUESTION.OR.MB$YESNO.OR.MB$DEFBUTTON1)
END
```

See Also

- M to N
- ABOUTBOXQQ
- SETMESSAGEQQ

Building Applications: Using QuickWin Overview

Building Applications: Displaying Message Boxes

MIN

Elemental Intrinsic Function (Generic):
Returns the minimum value of the arguments.

Syntax

`result = MIN (a1,a2[,a3...])`

a1, *a2*, *a3*	(Input) All must have the same type (integer or real) and kind parameters.

Results

For MIN0, AMIN1, DMIN1, QMIN1, IMIN0, JMIN0, and KMIN0, the result type is the same as the arguments. For MIN1, IMIN1, JMIN1, and KMIN1, the result type is integer. For AMIN0, AIMIN0, AJMIN0, and AKMIN0, the result type is real. The value of the result is that of the smallest argument.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	INTEGER(1)
IMIN0	INTEGER(2)	INTEGER(2)
AIMIN0	INTEGER(2)	REAL(4)
MIN0 [2]	INTEGER(4)	INTEGER(4)
AMIN0 [3, 4]	INTEGER(4)	REAL(4)
KMIN0	INTEGER(8)	INTEGER(8)
AKMIN0	INTEGER(8)	REAL(4)
IMIN1	REAL(4)	INTEGER(2)
MIN1 [4, 5, 6]	REAL(4)	INTEGER(4)
KMIN1	REAL(4)	INTEGER(8)
AMIN1 [7]	REAL(4)	REAL(4)
DMIN1	REAL(8)	REAL(8)
QMIN1	REAL(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

[2]Or JMIN0.

[3]Or AJMIN0.AMIN0 is the same as REAL (MIN).

Specific Name [1]	Argument Type	Result Type

[4]In Fortran 95/90, AMIN0 and MIN1 are specific functions with no generic name. For compatibility with older versions of Fortran, these functions can also be specified as generic functions.

[5]Or JMIN1.MIN1 is the same as INT (MIN).

[6]The setting of compiler options specifying integer size can affect MIN1.

[7]The setting of compiler options specifying real size can affect AMIN1.

Example

MIN (2.0, -8.0, 6.0) has the value -8.0.

MIN (14, 32, -50) has the value -50.

The following shows another example:

```
INTEGER m1, m2
REAL r1, r2
m1 = MIN (5, 6, 7)            ! returns 5
m2 = MIN1 (-5.7, 1.23, -3.8) ! returns -5
r1 = AMIN0 (-5, -6, -7)       ! returns -7.0
r2 = AMIN1(-5.7, 1.23, -3.8) ! returns -5.7
```

See Also

- M to N
- MAX

MINEXPONENT

Inquiry Intrinsic Function (Generic): *Returns the minimum exponent in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = MINEXPONENT (x)
```

x (Input) must be of type real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value e_{min}, as defined in Model for Real Data.

Example

If X is of type REAL(4), MINEXPONENT (X) has the value -125.

The following shows another example:

```
REAL(8) r1  ! DOUBLE PRECISION REAL

INTEGER i

i = MINEXPONENT (r1) ! returns - 1021.
```

See Also

- M to N
- MAXEXPONENT

MINLOC

Transformational Intrinsic Function (Generic):
Returns the location of the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MINLOC (array [, dim] [, mask] [, kind])`

array	(Input) Must be an array of type integer or real.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*. This argument is a Fortran 95 feature.
mask	(Input; optional) Must be a logical array that is conformable with *array*.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is an array of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The following rules apply if *dim* is omitted:

- The array result has rank one and a size equal to the rank of *array*.
- If MINLOC(*array*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value in *array*.

  The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of *array*.
- If MINLOC(*array*, MASK= *mask*) is specified, the elements in the array result form the subscript of the location of the element with the minimum value corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- The array result has a rank that is one less than *array*, and shape (d_1, d_2,...d_{dim-1}, d_{dim+1},...d_n), where (d_1, d_2,...d_n) is the shape of *array*.
- If *array* has rank one, MINLOC(*array*, *dim*[, *mask*]) has a value equal to that of MINLOC(*array*[,MASK = *mask*]). Otherwise, the value of element (s_1, s_2,...s_{dim-1}, s_{dim+1},...s_n) of MINLOC(*array*, *dim*[, *mask*]) is equal to MINLOC(*array*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n) [, MASK = *mask*(s_1, s_2,...s_{dim-1}, :, s_{dim+1},...s_n)]).

If more than one element has minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If *array* has size zero, or every element of *mask* has the value .FALSE., the value of the result is controlled by compiler option `assume [no]old_maxminloc`, which can set the result to either 1 or 0.

The setting of compiler options specifying integer size can affect this function.

Example

The value of MINLOC ((/3, 1, 4, 1/)) is (2), which is the subscript of the location of the first occurrence of the minimum value in the rank-one array.

A is the array

```
[  4    0   -3    2 ]
[  3    1   -2    6 ]
[ -1   -4    5   -5 ].
```

MINLOC (A, MASK=A .GT. -5) has the value (3, 2) because these are the subscripts of the location of the minimum value (-4) that is greater than -5.

MINLOC (A, DIM=1) has the value (3, 3, 1, 3). 3 is the subscript of the location of the minimum value (-1) in column 1; 3 is the subscript of the location of the minimum value (-4) in column 2; and so forth.

MINLOC (A, DIM=2) has the value (3, 3, 4). 3 is the subscript of the location of the minimum value (-3) in row 1; 3 is the subscript of the location of the minimum value (-2) in row 2; and so forth.

The following shows another example:

```
INTEGER i, minl(1)
INTEGER array(2, 3)
INTEGER, ALLOCATABLE :: AR1(:)
! put values in array
array = RESHAPE((/-7, 1, -2, -9, 5, 0/),(/2, 3/))
!   array is   -7 -2 5
!               1 -9 0
i = SIZE(SHAPE(array))  ! Get the number of dimensions
                        ! in array
ALLOCATE (AR1 (i) )     ! Allocate AR1 to number
                        ! of dimensions in array
AR1 = MINLOC (array, MASK = array .GT. -5) ! Get the
                        ! location (subscripts) of
                        ! smallest element greater
                        ! than -5 in array
!
! MASK = array .GT. -5 creates a mask array the same
! size and shape as array whose elements are .TRUE. if
! the corresponding element in array is greater than
! -5, and .FALSE. if it is not. This mask causes MINLOC
! to return the index of the element in array with the
! smallest value greater than -5.
!
!array is  -7 -2 5 and MASK= array .GT. -5 is  F T T
!           1 -9 0                             T F T
! and AR1 = MINLOC(array, MASK = array .GT. -5) returns
! (1, 2), the location of the element with value -2
minl = MINLOC((/-7,2,-7,5/))   ! returns 1, first
```

```
                                    ! occurrence of minimum
 END
```

See Also

- M to N
- MAXLOC
- MINVAL
- MAXVAL

MINVAL

Transformational Intrinsic Function (Generic):
Returns the minimum value of all elements in an array, a set of elements in an array, or elements in a specified dimension of an array.

Syntax

`result = MINVAL (array[,dim] [,mask])`

array	(Input) Must be an array of type integer or real.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be a logical array that is conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If MINVAL(*array*) is specified, the result has a value equal to the minimum value of all the elements in *array*.
- If MINVAL(*array*, MASK= *mask*) is specified, the result has a value equal to the minimum value of the elements in *array* corresponding to the condition specified by *mask*.

The following rules apply if *dim* is specified:

- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- If *array* has rank one, MINVAL(*array*, *dim*[, *mask*]) has a value equal to that of MINVAL(*array*[,MASK = *mask*]). Otherwise, the value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of MINVAL(*array*, *dim*, [, *mask*]) is equal to MINVAL(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

If *array* has size zero or if there are no true elements in *mask*, the result (if *dim* is omitted), or each element in the result array (if *dim* is specified), has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind parameters of *array*.

Example

The value of MINVAL ((/2, 3, 4/)) is 2 because that is the minimum value in the rank-one array.

The value of MINVAL (B, MASK=B .GT. 0.0) finds the minimum value of the positive elements of B.

C is the array

```
[ 2  3  4 ]
[ 5  6  7 ].
```

MINVAL (C, DIM=1) has the value (2, 3, 4). 2 is the minimum value in column 1; 3 is the minimum value in column 2; and so forth.

MINVAL (C, DIM=2) has the value (2, 5). 2 is the minimum value in row 1 and 5 is the minimum value in row 2.

The following shows another example:

```
INTEGER array(2, 3), i(2), minv
INTEGER, ALLOCATABLE :: AR1(:), AR2(:)
array = RESHAPE((/1, 4, 5, 2, 3, 6/), (/2, 3/))
!    array is    1 5 3
!                4 2 6
i = SHAPE(array)    ! i = [2 3]
ALLOCATE(AR1(i(2))) ! dimension AR1 to number of
                    ! elements in dimension 2
                    ! (a column) of array.
ALLOCATE(AR2(i(1))) ! dimension AR2 to number of
                    ! elements in dimension 1
                    ! (a row) of array
minv = MINVAL(array, MASK = array .GT. 4) ! returns 5
AR1 = MINVAL(array, DIM = 1)   ! returns [ 1 2 3 ]
AR2 = MINVAL(array, DIM = 2)   ! returns [ 1 2 ]
END
```

See Also

- M to N
- MAXVAL
- MINLOC
- MAXLOC

MM_PREFETCH

Intrinsic Subroutine (Generic): *Prefetches data from the specified address on one memory cache line. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

CALL MM_PREFETCH (*address*[,*hint*] [,*fault*] [,*exclusive*])

address (Input) Is the name of a scalar or array; it can be of any type or rank. It specifies the address of the data on the cache line to prefetch.

hint (Input; optional) Is an optional default integer constant with one of the following values:

Value	Prefetch Constant	Description
0	FOR_K_PREFETCH_T0	Prefetches into the L1 cache (and the L2 and the L3 cache). Use this for integer data.
1	FOR_K_PREFETCH_T1	Prefetches into the L2 cache (and the L3 cache); floating-point data is used from the L2 cache, not the L1 cache. Use this for real data.
2	FOR_K_PREFETCH_T2	Prefetches into the L2 cache (and the L3 cache); this line will be marked for early displacement. Use this if you are not going to reuse the cache line frequently.
3	FOR_K_PREFETCH_NTA	Prefetches into the L2 cache (but *not* the L3 cache); this line will be marked for early displacement. Use

Value	Prefetch Constant	Description
		this if you are not going to reuse the cache line.

The preceding constants are defined in file `fordef.for` on Windows* systems and file `fordef.f` on Linux* and Mac OS* X systems.
If *`hint`* is omitted, 0 is assumed.

`fault` (Input; optional) Is an optional default logical constant. If .TRUE. is specified, page faults are allowed to occur, if necessary; if .FALSE. is specified, page faults are not allowed to occur. If *`fault`* is omitted, .FALSE. is assumed. This argument is ignored on Intel® 64 architecture and IA-32 architecture.

`exclusive` (Input; optional) Is an optional default logical constant. If.TRUE. is specified, you get exclusive ownership of the cache line because you intend to assign to it; if .FALSE. is specified, there is no exclusive ownership. If *`exclusive`* is omitted, .FALSE.is assumed. This argument is ignored on Intel® 64 architecture and IA-32 architecture.

Example

```
subroutine spread_lf (a, b)
PARAMETER (n = 1025)
real*8 a(n,n), b(n,n), c(n)
do j = 1,n
  do i = 1,100
    a(i, j) = b(i-1, j) + b(i+1, j)
    call mm_prefetch (a(i+20, j), 1)
    call mm_prefetch (b(i+21, j), 1)
  enddo
enddo
print *, a(2, 567)
stop
end
```

MOD

Elemental Intrinsic Function (Generic):
Returns the remainder when the first argument is divided by the second argument.

Syntax

`result = MOD (a, p)`

a	(Input) Must be of type integer or real.
p	(Input)Must have the same type and kind parameters as *a*.

Results

The result type is the same as *a*. If *p* is not equal to zero, the value of the result is *a*- INT(*a*/ *p*) * *p*. If *p* is equal to zero, the result is undefined.

Specific Name	Argument Type	Result Type
BMOD	INTEGER(1)	INTEGER(1)

Specific Name	Argument Type	Result Type
IMOD[1]	INTEGER(2)	INTEGER(2)
MOD [2]	INTEGER(4)	INTEGER(4)
KMOD	INTEGER(8)	INTEGER(8)
AMOD [3]	REAL(4)	REAL(4)
DMOD [3,4]	REAL(8)	REAL(8)
QMOD	REAL(16)	REAL(16)

[1] Or HMOD.

[2] Or JMOD.

[3] The setting of compiler options specifying real size can affect AMOD and DMOD.

[4] The setting of compiler options specifying double size can affect DMOD.

Example

MOD (7, 3) has the value 1.

MOD (9, -6) has the value 3.

MOD (-9, 6) has the value -3.

The following shows more examples:

```
INTEGER I
REAL R
R = MOD(9.0, 2.0)  ! returns 1.0
I = MOD(18, 5)     ! returns 3
I = MOD(-18, 5)    ! returns -3
```

See Also

- M to N
- MODULO

MODIFYMENUFLAGSQQ (W*32, W*64)

QuickWin Function: *Modifies a menu item's state.*

Module

USE IFQWIN

Syntax

`result = MODIFYMENUFLAGSQQ (menuID,itemID,flag)`

menuID (Input) INTEGER(4). Identifies the menu containing the item whose state is to be modified, starting with 1 as the leftmost menu.

itemID (Input) INTEGER(4). Identifies the menu item whose state is to be modified, starting with 0 as the top item.

flags (Input) INTEGER(4). Constant indicating the menu state. Flags can be combined with an inclusive OR (see below). The following constants are available:

- $MENUGRAYED - Disables and grays out the menu item.
- $MENUDISABLED - Disables but does not gray out the menu item.
- $MENUENABLED - Enables the menu item.
- $MENUSEPARATOR - Draws a separator bar.
- $MENUCHECKED - Puts a check by the menu item.
- $MENUUNCHECKED - Removes the check by the menu item.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The constants available for flags can be combined with an inclusive OR where reasonable, for example $MENUCHECKED .OR. $MENUENABLED. Some combinations do not make sense, such as $MENUENABLED and $MENUDISABLED, and lead to undefined behavior.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
LOGICAL(4)    result
CHARACTER(20) str
! Append item to the bottom of the first (FILE) menu
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Gray out and disable the first two menu items in the
!  first (FILE) menu
result = MODIFYMENUFLAGSQQ (1, 1, $MENUGRAYED)
result = MODIFYMENUFLAGSQQ (1, 2, $MENUGRAYED)
END
```

See Also

- M to N
- APPENDMENUQQ
- DELETEMENUQQ
- INSERTMENUQQ
- MODIFYMENUROUTINEQQ
- MODIFYMENUSTRINGQQ

Building Applications: Using QuickWin Overview

Building Applications: Program Control of Menus

MODIFYMENUROUTINEQQ (W*32, W*64)

QuickWin Function: *Changes a menu item's callback routine.*

Module

USE IFQWIN

Syntax

```
result = MODIFYMENUROUTINEQQ (menuIdD,itemID,routine)
```

menuID (Input) INTEGER(4). Identifies the menu that contains the item whose callback routine is be changed, starting with 1 as the leftmost menu.

itemID (Input) INTEGER(4). Identifies the menu item whose callback routine is to be changed, starting with 0 as the top item.

routine (Input) EXTERNAL. Callback subroutine called if the menu item is selected. All routines take a single LOGICAL parameter that indicates whether the menu item is checked or not. You can assign the following predefined routines to menus:

- WINPRINT - Prints the program.
- WINSAVE - Saves the program.
- WINEXIT - Terminates the program.
- WINSELECTTEXT - Selects text from the current window.
- WINSELECTGRAPHICS - Selects graphics from the current window.
- WINSELECTALL - Selects the entire contents of the current window.
- WININPUT - Brings to the top the child window requesting input and makes it the current window.
- WINCOPY - Copies the selected text and/or graphics from the current window to the Clipboard.
- WINPASTE - Allows the user to paste Clipboard contents (text only) to the current text window of the active window during a READ.
- WINCLEARPASTE - Clears the paste buffer.
- WINSIZETOFIT - Sizes output to fit window.
- WINFULLSCREEN - Displays output in full screen.
- WINSTATE - Toggles between pause and resume states of text output.
- WINCASCADE - Cascades active windows.
- WINTILE - Tiles active windows.
- WINARRANGE - Arranges icons.
- WINSTATUS - Enables a status bar.

- WININDEX - Displays the index for QuickWin help.
- WINUSING - Displays information on how to use Help.
- WINABOUT - Displays information about the current QuickWin application.
- NUL - No callback routine.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- M to N
- APPENDMENUQQ
- DELETEMENUQQ
- INSERTMENUQQ
- MODIFYMENUFLAGSQQ
- MODIFYMENUSTRINGQQ

Building Applications: Using QuickWin Overview

Building Applications: Program Control of Menus

MODIFYMENUSTRINGQQ (W*32, W*64)

QuickWin Function: *Changes a menu item's text string.*

Module

USE IFQWIN

Syntax

```
result = MODIFYMENUSTRINGQQ (menuID,itemID,text)
```

menuID (Input) INTEGER(4). Identifies the menu containing the item whose text string is to be changed, starting with 1 as the leftmost item.

itemID	(Input) INTEGER(4). Identifies the menu item whose text string is to be changed, starting with 0 as the top menu item.
text	(Input) Character*(*). Menu item name. Must be a null-terminated C string. For example, words of text'C.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can add access keys in your text strings by placing an ampersand (&) before the letter you want underlined. For example, to add a Print menu item with the r underlined, use "P&rint"C as *text*.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
LOGICAL(4) result
CHARACTER(25) str
! Append item to the bottom of the first (FILE) menu
str = '&Add to File Menu'C
result = APPENDMENUQQ(1, $MENUENABLED, str, WINSTATUS)
! Change the name of the first item in the first menu
str ='&Browse'C
result = MODIFYMENUSTRINGQQ (1, 1, str)
END
```

See Also

- M to N
- APPENDMENUQQ
- DELETEMENUQQ
- INSERTMENUQQ
- SETMESSAGEQQ
- MODIFYMENUFLAGSQQ

- MODIFYMENUROUTINEQQ

Building Applications: Using QuickWin Overview

Building Applications: Program Control of Menus

Building Applications: Displaying Character-Based Text

MODULE

Statement: *Marks the beginning of a module program unit, which contains specifications and definitions that can be used by one or more program units.*

Syntax

```
MODULE name
   [specification-part]
[CONTAINS
   module-subprogram
   [module-subprogram]...]
END[ MODULE [name]]
```

name Is the name of the module.

specification-part Is one or more specification statements, except for the following:

- ENTRY
- FORMAT
- AUTOMATIC (or its equivalent attribute)
- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- Statement functions

An automatic object must not appear in a specification statement.

module-subprogram Is a function or subroutine subprogram that defines the module procedure. A function must end with END FUNCTION and a subroutine must end with END SUBROUTINE.
A module subprogram can contain internal procedures.

Description

If a name follows the END statement, it must be the same as the name specified in the MODULE statement.

The module name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A module is host to any module procedures it contains, and entities in the module are accessible to the module procedures through host association.

A module must not reference itself (either directly or indirectly).

You can use the PRIVATE attribute to restrict access to procedures or variables within a module.

Although ENTRY statements, FORMAT statements, and statement functions are not allowed in the specification part of a module, they are allowed in the specification part of a module subprogram.

Any executable statements in a module can only be specified in a module subprogram.

A module can contain one or more procedure interface blocks, which let you specify an explicit interface for an external subprogram or dummy subprogram.

Example

The following example shows a simple module that can be used to provide global data:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5)
END MODULE MOD_A
...
SUBROUTINE SUB_Z
  USE MOD_A               ! Makes scalar variables B and C, and array
  ...                     !   E available to this subroutine
END SUBROUTINE SUB_Z
```

The following example shows a module procedure:

```
MODULE RESULTS
...
CONTAINS
  FUNCTION MOD_RESULTS(X,Y)  ! A module procedure
  ...
  END FUNCTION MOD_RESULTS
END MODULE RESULTS
```

The following example shows a module containing a derived type:

```
MODULE EMPLOYEE_DATA
  TYPE EMPLOYEE
    INTEGER ID
    CHARACTER(LEN=40) NAME
  END TYPE EMPLOYEE
END MODULE
```

The following example shows a module containing an interface block:

```
MODULE ARRAY_CALCULATOR
  INTERFACE
    FUNCTION CALC_AVERAGE(D)
      REAL :: CALC_AVERAGE
      REAL, INTENT(IN) :: D(:)
    END FUNCTION
  END INTERFACE
END MODULE ARRAY_CALCULATOR
```

The following example shows a derived-type definition that is public with components that are private:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

This design allows you to change components of a type without affecting other program units that use the module.

If a derived type is needed in more than one program unit, the definition should be placed in a module and accessed by a USE statement whenever it is needed, as follows:

```
MODULE STUDENTS
  TYPE STUDENT_RECORD
  ...
  END TYPE
CONTAINS
  SUBROUTINE COURSE_GRADE(...)
  TYPE(STUDENT_RECORD) NAME
  ...
  END SUBROUTINE
END MODULE STUDENTS
...
PROGRAM SENIOR_CLASS
  USE STUDENTS
  TYPE(STUDENT_RECORD) ID
  ...
END PROGRAM
```

Program SENIOR_CLASS has access to type STUDENT_RECORD, because it uses module STUDENTS. Module procedure COURSE_GRADE also has access to type STUDENT_RECORD, because the derived-type definition appears in its host.

The following shows another example:

```
MODULE mod1
  REAL(8) a,b,c,d
  INTEGER(4) Int1, Int2, Int3
CONTAINS
  function fun1(x)
....
  end function fun1
END MODULE
```

See Also

- M to N
- PUBLIC
- PRIVATE
- USE
- Procedure Interfaces
- Program Units and Procedures
- PROTECTED Attribute and Statement

MODULE PROCEDURE

Statement: *Identifies module procedures in an interface block that specifies a generic name. See INTERFACE.*

Example

```
!A program that changes non-default integers and reals !into default integers and reals
   PROGRAM CHANGE_KIND
   USE Module1
   INTERFACE DEFAULT
     MODULE PROCEDURE Sub1, Sub2
   END INTERFACE
   integer(2) in
   integer indef
   indef = DEFAULT(in)
   END PROGRAM
! procedures sub1 and sub2 defined as follows:
    MODULE Module1
    CONTAINS
      FUNCTION Sub1(y)
        REAL(8) y
        sub1 = REAL(y)
      END FUNCTION
      FUNCTION Sub2(z)
        INTEGER(2) z
        sub2 = INT(z)
      END FUNCTION
    END MODULE
```

See Also

- M to N
- MODULE
- Modules and Module Procedures

MODULO

Elemental Intrinsic Function (Generic):
Returns the modulo of the arguments.

Syntax

```
result = MODULO (a,p)
```

a	(Input) Must be of type integer or real.
p	(Input) Must have the same type and kind parameters as *a*.

Results

The result type is the same *a*. The result value depends on the type of *a*, as follows:

- If *a* is of type integer and P is not equal to zero, the value of the result is *a* - FLOOR(REAL(*a*) / REAL(*p*)) * *p*.
- If *a* is of type real and *p* is not equal to zero, the value of the result is *a* - FLOOR(*a*/ *p*) * *p*.

If *p* is equal to zero (regardless of the type of *a*), the result is undefined.

Example

MODULO (7, 3) has the value 1.

MODULO (9, -6) has the value -3.

MODULO (-9, 6) has the value 3.

The following shows more examples:

```
INTEGER I

REAL R

I= MODULO(8, 5)          ! returns 3          Note: q=1

I= MODULO(-8, 5)         ! returns 2          Note: q=-2

I= MODULO(8, -5)         ! returns -2         Note: q=-2

R= MODULO(7.285, 2.35)   ! returns 0.2350001  Note: q=3

R= MODULO(7.285, -2.35) ! returns -2.115     Note: q=-4
```

See Also

- M to N
- MOD

MOVE_ALLOC

Intrinsic Subroutine (Generic): *Moves an allocation from one allocatable object to another. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL MOVE_ALLOC (from,to)
```

from	(Input; output) Can be of any type and rank; it must be allocatable.
to	(Output) Must have the same type and kind parameters as *from* and have the same rank; it must be allocatable.

If *to* is currently allocated, it is deallocated. If *from* is allocated, *to* becomes allocated with the same type, type parameters, array bounds, and value as *from*. Lastly, *from* is deallocated.

If *to* has the TARGET attribute, any pointer associated with *from* at the time of the call to MOVE_ALLOC becomes correspondingly associated with *to*. If *to* does not have the TARGET attribute, the pointer association status of any pointer associated with *from* on entry becomes undefined.

During implementation of MOVE_ALLOC, the internal descriptor contents are copied from *from* to *to*, so that the storage pointed to is the same.

Typically, MOVE_ALLOC is used to provide an efficient way to reallocate a variable to a larger size without copying the data twice.

Example

The following shows an example of how to increase the allocated size of A and keep the old values with only one copy of the old values.

```
integer,allocatable::a(:),b(:)
n=2
allocate (a(n), b(n*2))
a=(/(i,i=1,n)/)
b=-1
print *, ' Old a = ',a
print *, ' Old b = ',b
print *, ' Allocated(a), allocated(b) = ', allocated(a), allocated(b)
b(1:n)=a   ! Copy all of a into low end of b (the only copy)
print *, ' New b = ',b
call move_alloc(b,a) ! Make a the container, deallocate b (NO copy!)
print *, ' New a = ',a
print *, ' Allocated(a), allocated(b) = ', allocated(a), allocated(b)
end
```

The following shows another example:

```
! This program uses MOVE_ALLOC to make an allocated array X bigger and
! keep the old values of X by only making one copy of the old values of X
    integer :: n = 2
    real, allocatable :: x(:), y(:)
    allocate (x(n), y(2*n))         ! Y is bigger than X
    x = (/(i,i=1,n)/)       ! put "old values" into X
    Y = -1                  ! put different "old values" into Y
    print *, ' allocated of X is ', allocated (X)
    print *, ' allocated of Y is ', allocated (Y)
    print *, ' old X is ', X
    print *, ' old Y is ', Y
    y (1:n) = x             ! copy all of X into the first locations of Y
                            ! this is the only copying of values required
    print *, ' new Y is ', y
    call move_alloc (y, x)  ! X is now twice a big as it was, Y is
                            ! deallocated, the values were not copied
    print *, ' allocated of X is ', allocated (X)
    print *, ' allocated of Y is ', allocated (Y)
    print *, ' new X is ', x
    end
```

The following shows the output for the above example:

```
 allocated of X is  T
 allocated of Y is  T
 old X is    1.000000       2.000000
 old Y is   -1.000000      -1.000000      -1.000000      -1.000000
 new Y is    1.000000       2.000000      -1.000000      -1.000000
 allocated of X is  T
 allocated of Y is  F
 new X is    1.000000       2.000000      -1.000000      -1.000000
```

MOVETO, MOVETO_W (W*32, W*64)

Graphics Subroutines: *Move the current graphics position to a specified point. No drawing occurs.*

Module

USE IFQWIN

Syntax

`CALL MOVETO (x,y,t)`

`CALL MOVETO_W (wx,wy,wt)`

x, y	(Input) INTEGER(2). Viewport coordinates of the new graphics position.

t	(Output) Derived type `xycoord`. Viewport coordinates of the previous graphics position. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:
	`TYPE xycoord`
	`INTEGER(2) xcoord ! x coordinate`
	`INTEGER(2) ycoord ! y coordinate`
	`END TYPE xycoord`

wx, wy	(Input) REAL(8). Window coordinates of the new graphics position.

wt (Output) Derived type wxycoord. Window coordinates of the previous graphics position. The derived type wxycoord is defined in IFQWIN.F90

```
TYPE wxycoord
  REAL(8) wx  ! x window coordinate
  REAL(8) wy  ! y window coordinate
END TYPE wxycoord
```

MOVETO sets the current graphics position to the viewport coordinate (*x*, *y*). MOVETO_W sets the current graphics position to the window coordinate (*wx*, *wy*).

MOVETO and MOVETO_W assign the coordinates of the previous position to *t* and *wt*, respectively.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics ap.
USE IFQWIN
INTEGER(2) status, x, y
INTEGER(4) result
TYPE (xycoord) xy
RESULT = SETCOLORRGB(Z'FF0000') ! blue
x = 60
! Draw a series of lines
DO y = 50, 92, 3
  CALL MOVETO(x, y, xy)
  status = LINETO(INT2(x + 20), y)
END DO
END
```

See Also

- M to N
- GETCURRENTPOSITION
- LINETO
- OUTGTEXT

Building Applications: Drawing Lines on the Screen

Building Applications: SHOWFONT.F90 Example

Building Applications: Using Fonts from the Graphics Library Overview

MULT_HIGH (i64 only)

Elemental Intrinsic Function (Specific):
Multiplies two 64-bit unsigned integers. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax

```
result = MULT_HIGH (i,j)
```

i (Input) Must be of type INTEGER(8).

j (Input) Must be of type INTEGER(8).

Results

The result type is INTEGER(8). The result value is the upper (leftmost) 64 bits of the 128-bit unsigned result.

Example

Consider the following:

```
          INTEGER(8) I,J,K
          I=2_8**53
          J=2_8**51
          K = MULT_HIGH (I,J)
          PRINT *,I,J,K
          WRITE (6,1000)I,J,K
  1000    FORMAT (' ', 3(Z,1X))
          END
```

This example prints the following:

```
 9007199254740992      2251799813685248         1099511627776
   20000000000000           8000000000000             10000000000
```

MULT_HIGH_SIGNED (i64 only)

Elemental Intrinsic Function (Specific):
Multiplies two 64-bit signed integers. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax

```
result = MULT_HIGH_SIGNED (i,j)
```

i (Input) Must be of type INTEGER(8).

j (Input) Must be of type INTEGER(8).

Results

The result type is INTEGER(8). The result value is the upper (leftmost) 64 bits of the 128-bit signed result.

Example

Consider the following:

```
          INTEGER(8) I,J,K
          I=2_8**53
          J=2_8**51
          K = MULT_HIGH_SIGNED (I,J)
          PRINT *,I,J,K
          WRITE (6,1000)I,J,K
  1000    FORMAT (' ', 3(Z,1X))
          END
```

This example prints the following:

```
  9007199254740992     -2251799813685248        -1099511627776
    20000000000000        FFF8000000000000        FFFFFF0000000000
```

MVBITS

Elemental Intrinsic Subroutine (Generic):
Copies a sequence of bits (a bit field) from one location to another. Intrinsic subroutines cannot be passed as actual arguments.

Syntax

`CALL MVBITS (from,frompos,len,to,topos)`

from	(Input) Integer. Can be of any integer type. It represents the location from which a bit field is transferred.
frompos	(Input) Can be of any integer type; it must not be negative. It identifies the first bit position in the field transferred from *from*. *frompos* + *len* must be less than or equal to BIT_SIZE(*from*).
len	(Input) Can be of any integer type; it must not be negative. It identifies the length of the field transferred from *from*.

to	(Input; output) Can be of any integer type, but must have the same kind parameter as *from*. It represents the location to which a bit field is transferred. *to* is set by copying the sequence of bits of length *len*, starting at position *frompos* of *from* to position *topos* of *to*. No other bits of *to* are altered.
topos	(Input) Can be of any integer type; it must not be negative. It identifies the starting position (within *to*) for the bits being transferred. *topos* + *len* must be less than or equal to BIT_SIZE(*to*).

For more information on bit functions, see Bit Functions.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

You can also use the following specific routines:

BMVBITS	Arguments *from* and *to* must be INTEGER(1).
HMVBITS	Arguments *from* and *to* must be INTEGER(2).
IMVBITS	Arguments *from* and *to* must be INTEGER(2).
JMVBITS	Arguments *from* and *to* must be INTEGER(4).
KMVBITS	Arguments *from* and *to* must be INTEGER(8).

Example

If TO has the initial value of 6, its value after a call to MVBITS(7, 2, 2, TO, 0) is 5.

The following shows another example:

```
INTEGER(1) :: from = 13  ! 00001101
INTEGER(1) :: to = 6     ! 00000110
CALL MVBITS(from, 2, 2, to, 0) ! returns to = 00000111
END
```

See Also

- M to N
- BIT_SIZE

- IBCLR
- IBSET
- ISHFT
- ISHFTC

NAMELIST

Statement: *Associates a name with a list of variables. This group name can be referenced in some input/output operations.*

Syntax

NAMELIST */group/ var-list*[[,] */group/ var-list*]...

group	Is the name of the group.
var-list	Is a list of variables (separated by commas) that are to be associated with the preceding group name. The variables can be of any data type.

Description

The namelist group name is used by namelist I/O statements instead of an I/O list. The unique group name identifies a list whose entities can be modified or transferred.

A variable can appear in more than one namelist group.

Each variable in *var-list* must be accessed by use or host association, or it must have its type, type parameters, and shape explicitly or implicitly specified in the same scoping unit. If the variable is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The following variables cannot be specified in a namelist group:

- An array dummy argument with nonconstant bounds
- A variable with assumed character length
- An allocatable array
- A pointer
- A variable of a type that has a pointer as an ultimate component
- A subobject of any of the above objects

Only the variables specified in the namelist can be read or written in namelist I/O. It is not necessary for the input records in a namelist input statement to define every variable in the associated namelist.

The order of variables in the namelist controls the order in which the values appear on namelist output. Input of namelist values can be in any order.

If the group name has the PUBLIC attribute, no item in the variable list can have the PRIVATE attribute.

The group name can be specified in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the group name is treated as a continuation of the list for that group name.

Example

In the following example, D and E are added to the variables A, B, and C for group name LIST:

```
NAMELIST /LIST/ A, B, C

NAMELIST /LIST/ D, E
```

In the following example, two group names are defined:

```
CHARACTER*30 NAME(25)

NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

Group name INPUT contains variables NAME, GRADE, and DATE. Group name OUTPUT contains variables TOTAL and NAME.

The following shows another example:

```
NAMELIST /example/ i1, l1, r4, r8, z8, z16, c1, c10, iarray
! The corresponding input statements could be:
&example
i1   = 11
l1   = .TRUE.
r4   = 24.0
r8   = 28.0d0
z8   = (38.0, 0.0)
z16 = (316.0d0, 0.0d0)
c1   = 'A'
c10 = 'abcdefghij'
iarray(8) = 41, 42, 43
/
```

A sample program, NAMELIST.F90, is included in the `<install-dir>/samples` subdirectory.

See Also

- M to N
- READ
- WRITE
- Namelist Specifier
- Namelist Input
- Namelist Output

NARGS

Inquiry Intrinsic Function (Specific): *Returns the total number of command-line arguments, including the command. This function cannot be passed as an actual argument.*

Syntax

```
result = NARGS( )
```

Results

The result type is INTEGER(4). The result is the number of command-line arguments, including the command. For example, NARGS returns 4 for the command-line invocation of `PROG1 -g -c -a`.

Example

```
    INTEGER(2) result
    result = RUNQQ('myprog', '-c -r')
    END
!  MYPROG.F90 responds to command switches -r, -c,
!  and/or -d
INTEGER(4) count, num, i, status
CHARACTER(80) buf
REAL r1 / 0.0 /
COMPLEX c1 / (0.0,0.0) /
REAL(8) d1 / 0.0 /
num = 5
count = NARGS()
DO i = 1, count-1
   CALL GETARG(i, buf, status)
   IF (status .lt. 0) THEN
     WRITE (*,*) 'GETARG error - exiting'
     EXIT
   END IF
   IF (buf(2:status) .EQ.'r') THEN
     r1 = REAL(num)
     WRITE (*,*) 'r1 = ',r1
   ELSE IF (buf(2:status) .EQ.'c') THEN
     c1 = CMPLX(num)
     WRITE (*,*) 'c1 = ', c1
   ELSE IF (buf(2:status) .EQ.'d') THEN
     d1 = DBLE(num)
     WRITE (*,*) 'd1 = ', d1
```

```
    ELSE
      WRITE(*,*) 'Invalid command switch: ', buf (1:status)
    END IF
 END DO
 END
```

See Also

- M to N
- GETARG
- IARGC
- COMMAND_ARGUMENT_COUNT
- GET_COMMAND
- GET_COMMAND_ARGUMENT

NEAREST

Elemental Intrinsic Function (Generic):
Returns the nearest different number (representable on the processor) in a given direction.

Syntax

`result = NEAREST (x, s)`

x	(Input) Must be of type real.
s	(Input) Must be of type real and nonzero.

Results

The result type is the same as *x*. The result has a value equal to the machine representable number that is different from and nearest to *x*, in the direction of the infinity with the same sign as *s*.

Example

If 3.0 and 2.0 are REAL(4) values, NEAREST (3.0, 2.0) has the value $3 + 2^{-22}$, which equals approximately 3.0000002. (For more information on the model for REAL(4), see Model for Real Data.

The following shows another example:

```
REAL(4) r1
REAL(8) r2, result
r1 = 3.0
result = NEAREST (r1, -2.0)
WRITE(*,*) result           ! writes 2.999999761581421
! When finding nearest to REAL(8), can't see
! the difference unless output in HEX
r2 = 111502.07D0
result = NEAREST(r2, 2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851ED
result = NEAREST(r2, -2.0)
WRITE(*,'(1x,Z16)') result  ! writes 40FB38E11EB851EB
END
```

See Also

- M to N
- EPSILON

NEW_LINE

Inquiry Intrinsic Function (Generic): *Returns a new line character.*

Syntax

`result = NEW_LINE(a)`

a	(Input) Must be of type default character. It may be a scalar or an array.

Results

The result is a character scalar of length one with the same kind type parameter as a.

The result value is the ASCII newline character ACHAR(10).

NINT

Elemental Intrinsic Function (Generic):
Returns the nearest integer to the argument.

Syntax

```
result = NINT (a[,kind])
```

a	(Input) Must be of type real.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *a* is greater than zero, NINT(*a*) has the value INT(*a*+ 0.5); if *a* is less than or equal to zero, NINT(*a*) has the value INT(*a*- 0.5).

Specific Name	Argument Type	Result Type
ININT	REAL(4)	INTEGER(2)
NINT [1, 2]	REAL(4)	INTEGER(4)
KNINT	REAL(4)	INTEGER(8)
IIDNNT	REAL(8)	INTEGER(2)
IDNINT [2, 3]	REAL(8)	INTEGER(4)
KIDNNT	REAL(8)	INTEGER(8)
IIQNNT	REAL(16)	INTEGER(2)
IQNINT[2, 4]	REAL(16)	INTEGER(4)
KIQNNT	REAL(16)	INTEGER(8)

[1]Or JNINT.

Specific Name	Argument Type	Result Type

[2]The setting of compiler options specifying integer size can affect NINT, IDNINT, and IQNINT.

[3]Or JIDNNT. For compatibility with older versions of Fortran, IDNINT can also be specified as a generic function.

[4]Or JIQNNT. For compatibility with older versions of Fortran, IQNINT can also be specified as a generic function.

Example

NINT (3.879) has the value 4.

NINT (-2.789) has the value -3.

The following shows another example:

```
INTEGER(4) i1, i2

i1 = NINT(2.783) ! returns 3

i2 = IDNINT(-2.783D0) ! returns -3
```

See Also

- M to N
- ANINT
- INT

NLSEnumCodepages (W*32, W*64)

NLS Function: *Returns an array containing the codepages supported by the system, with each array element describing one valid codepage.*

Module

USE IFNLS

Syntax

ptr=> NLSEnumCodepages( )

Results

The result is a pointer to an array of codepages, with each element describing one supported codepage.

NOTE. After use, the pointer returned by NLSEnumCodepages should be deallocated with the DEALLOCATE statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- NLSEnumLocales
- DEALLOCATE

Building Applications: Locale Setting and Inquiry Routines

NLSEnumLocales (W*32, W*64)

NLS Function: *Returns an array containing the language and country combinations supported by the system, in which each array element describes one valid combination.*

Module

USE IFNLS

Syntax

```
ptr=> NLSEnumLocales( )
```

Results

The result is a pointer to an array of locales, in which each array element describes one supported language and country combination. Each element has the following structure:

```
TYPE NLS$EnumLocale
  CHARACTER*(NLS$MaxLanguageLen)  Language
  CHARACTER*(NLS$MaxCountryLen)   Country
  INTEGER(4)                      DefaultWindowsCodepage
  INTEGER(4)                      DefaultConsoleCodepage
END TYPE
```

If the application is a Windows or QuickWin application, NLS$DefaultWindowsCodepage is the codepage used by default for the given language and country combination. If the application is a console application, NLS$DefaultConsoleCodepage is the codepage used by default for the given language and country combination.

NOTE. After use, the pointer returned by NLSEnumLocales should be deallocated with the DEALLOCATE statement.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- NLSEnumCodepages
- DEALLOCATE

Building Applications: Locale Setting and Inquiry Routines

NLSFormatCurrency (W*32, W*64)

NLS Function: *Returns a correctly formatted currency string for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatCurrency (outstr,instr[,flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted currency for the current locale. If *outstr* is longer than the formatted currency, it is blank-padded.
intstr	(Input) Character*(*). Number string to be formatted. Can contain only the characters '0' through '9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.

flags (Input; optional) INTEGER(4). If specified, modifies the currency conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are:

- NLS$Normal - No special formatting
- NLS$NoUserOverride - Do not use user overrides

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* (bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *instr* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
CHARACTER(40) str
INTEGER(4) i
i = NLSFormatCurrency(str, "1.23")
print *, str                              ! prints $1.23
i = NLSFormatCurrency(str, "1000000.99")
print *, str                              ! prints $1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatCurrency(str, "1.23")
print *, str                              ! prints 1 Pts
i = NLSFormatCurrency(str, "1000000.99")
print *, str                              ! prints 1.000.001 Pts
```

See Also

- M to N
- NLSFormatNumber
- NLSFormatDate
- NLSFormatTime

Building Applications: NLS Formatting Routines

NLSFormatDate (W*32, W*64)

NLS Function: *Returns a correctly formatted string containing the date for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatDate (outstr[,intime] [,flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted date for the current locale. If *outstr* is longer than the formatted date, it is blank-padded.
intime	(Input; optional) INTEGER(4). If specified, date to be formatted for the current locale. Must be an integer date such as the packed time created with PACKTIMEQQ. If you omit *intime*, the current system date is formatted and returned in *outstr*.
flags	(Input; optional) INTEGER(4). If specified, modifies the date conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides • NLS$UseAltCalendar - Use the locale's alternate calendar • NLS$LongDate - Use local long date format • NLS$ShortDate - Use local short date format

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr*(bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *intime* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
INTEGER(4) i
CHARACTER(40) str
i = NLSFORMATDATE(str, NLS$NORMAL)           ! 8/1/99
i = NLSFORMATDATE(str, NLS$USEALTCALENDAR)   ! 8/1/99
i = NLSFORMATDATE(str, NLS$LONGDATE)         ! Monday, August 1, 1999
i = NLSFORMATDATE(str, NLS$SHORTDATE)        ! 8/1/99
END
```

See Also

- M to N
- NLSFormatTime
- NLSFormatCurrency
- NLSFormatNumber

Building Applications: NLS Formatting Routines

NLSFormatNumber (W*32, W*64)

NLS Function: *Returns a correctly formatted number string for the current locale.*

Module

USE IFNLS

Syntax

`result = NLSFormatNumber (outstr,instr[,flags])`

outstr	(Output) Character*(*). String containing the correctly formatted number for the current locale. If *outstr* is longer than the formatted number, it is blank-padded.
instr	(Input) Character*(*). Number string to be formatted. Can only contain the characters '0' through '9', one decimal point (a period) if a floating-point value, and a minus sign in the first position if negative. All other characters are invalid and cause the function to return an error.
flags	(Input; optional) INTEGER(4). If specified, modifies the number conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are:A BR tag was used here in the original source.A BR tag was used here in the original source. • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr*(bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *instr* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
CHARACTER(40) str
INTEGER(4) i
i = NLSFormatNumber(str, "1.23")
print *, str                            ! prints 1.23
i = NLSFormatNumber(str, "1000000.99")
print *, str                            ! prints 1,000,000.99
i = NLSSetLocale("Spanish", "Spain")
i = NLSFormatNumber(str, "1.23")
print *, str                            ! prints 1,23
i = NLSFormatNumber(str, "1000000.99")
print *, str                            ! prints 1.000.000,99
```

See Also

- M to N
- NLSFormatTime
- NLSFormatCurrency
- NLSFormatDate

NLSFormatTime (W*32, W*64)

NLS Function: *Returns a correctly formatted string containing the time for the current locale.*

Module

USE IFNLS

Syntax

```
result = NLSFormatTime (outstr [,intime] [,flags])
```

outstr	(Output) Character*(*). String containing the correctly formatted time for the current locale. If *outstr* is longer than the formatted time, it is blank-padded.
intime	(Input; optional) INTEGER(4). If specified, time to be formatted for the current locale. Must be an integer time such as the packed time created with PACKTIMEQQ. If you omit *intime*, the current system time is formatted and returned in outstr.
flags	(Input; optional) INTEGER(4). If specified, modifies the time conversion. If you omit *flags*, the flag NLS$Normal is used. Available values (defined in IFNLS.F90) are: • NLS$Normal - No special formatting • NLS$NoUserOverride - Do not use user overrides • NLS$NoMinutesOrSeconds - Do not return minutes or seconds • NLS$NoSeconds - Do not return seconds • NLS$NoTimeMarker - Do not add a time marker string • NLS$Force24HourFormat - Return string in 24 hour format

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr*(bytes are counted, not multibyte characters). If an error occurs, the result is one of the following negative values:

- NLS$ErrorInsufficentBuffer - *outstr* buffer is too small
- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorInvalidInput - *intime* has an illegal value

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
INTEGER(4) i
CHARACTER(20) str
i = NLSFORMATTIME(str, NLS$NORMAL)              ! 11:38:28 PM
i = NLSFORMATTIME(str, NLS$NOMINUTESORSECONDS)  ! 11 PM
i = NLSFORMATTIME(str, NLS$NOTIMEMARKER)        ! 11:38:28 PM
i = NLSFORMATTIME(str, IOR(NLS$FORCE24HOURFORMAT,               &
&                      NLS$NOSECONDS))          ! 23:38 PM
END
```

See Also

- M to N
- NLSFormatCurrency
- NLSFormatDate
- NLSFormatNumber

Building Applications: NLS Formatting Routines

NLSGetEnvironmentCodepage (W*32, W*64)

NLS Function: *Returns the codepage number for the system (Window) codepage or the console codepage.*

Module

USE IFNLS

Syntax

`result = NLSGetEnvironmentCodepage (flags)`

flags	(Input) INTEGER(4). Tells the function which codepage number to return. Available values (defined in `IFNLS.F90`) are: • NLS$ConsoleEnvironmentCodepage - Gets the codepage for the console

- NLS$WindowsEnvironmentCodepage - Gets the current Windows codepage

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, it returns one of the following error codes:

- NLS$ErrorInvalidFlags - *flags* has an illegal value
- NLS$ErrorNoConsole - There is no console associated with the given application; so, operations with the console codepage are not possible

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- NLSSetEnvironmentCodepage

Building Applications: Locale Setting and Inquiry Routines

NLSGetLocale (W*32, W*64)

NLS Subroutine: *Returns the current language, country, or codepage.*

Module

USE IFNLS

Syntax

```
CALL NLSGetLocale ([language] [,country] [,codepage])
```

language	(Output; optional) Character*(*). Current language.
country	(Output; optional) Character*(*). Current country.
codepage	(Output; optional) INTEGER(4). Current codepage.

NLSGetLocale returns a valid codepage in *codepage*. It does not return one of the NLS$... symbolic constants that can be used with NLSSetLocale.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
CHARACTER(50) cntry, lang
INTEGER(4)    code
CALL NLSGetLocale (lang, cntry, code)    ! get all three
CALL NLSGetLocale (CODEPAGE = code)      ! get the codepage
CALL NLSGetLocale (COUNTRY = cntry, CODEPAGE =code) ! get country
                                                    ! and codepage
```

See Also

- M to N
- NLSSetLocale

Building Applications: Locale Setting and Inquiry Routines

NLSGetLocaleInfo (W*32, W*64)

NLS Function: *Returns information about the current locale.*

Module

USE IFNLS

Syntax

`result = NLSGetLocaleInfo (type,outstr)`

type	(Input) INTEGER(4). NLS parameter requested. A list of parameter names is provided in NLS LocaleInfo Parameters.
outstr	(Output) Character*(*). Parameter setting for the current locale. All parameter settings placed in *outstr* are character strings, even numbers. If a parameter setting is numeric, the ASCII representation of the number is used. If the requested parameter is a date or time string, an explanation of how to interpret the format in *outstr* is provided in NLS Date and Time Format.

Results

The result type is INTEGER(4). The result is the number of characters written to *outstr* if successful, or if *outstr* has 0 length, the number of characters required to hold the requested information. Otherwise, the result is one of the following error codes (defined in IFNLS.F90):

- NLS$ErrorInvalidLIType - The given *type* is invalid
- NLS$ErrorInsufficientBuffer - The *outstr* buffer was too small, but was not 0 (so that the needed size would be returned)

The NLS$LI parameters are used for the argument *type* and select the locale information returned by NLSGetLocaleInfo in *outstr*. You can perform an inclusive OR with NLS$NoUserOverride and any NLS$LI parameter. This causes NLSGetLocaleInfo to bypass any user overrides and always return the system default value.

The following table lists and describes the NLS$LI parameters.

Table 856: NLS LocaleInfo Parameters

Parameter	Description
NLS$LI_ILANGUAGE	An ID indicating the language.
NLS$LI_SLANGUAGE	The full localized name of the language.
NLS$LI_SENGLANGUAGE	The full English name of the language from the ISO Standard 639. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SABBREVLANGNAME	The abbreviated name of the language, created by taking the 2-letter language abbreviation as found in ISO Standard 639 and adding a third letter as appropriate to indicate the sublanguage.
NLS$LI_SNATIVELANGNAME	The native name of the language.
NLS$LI_ICOUNTRY	The country code, based on international phone codes, also referred to as IBM country codes.
NLS$LI_SCOUNTRY	The full localized name of the country.

Parameter	Description
NLS$LI_SENGCOUNTRY	The full English name of the country. This will always be restricted to characters that map into the ASCII 127 character subset.
NLS$LI_SABBREVCTRYNAME	The abbreviated name of the country as per ISO Standard 3166.
NLS$LI_SNATIVECTRYNAME	The native name of the country.
NLS$LI_IDEFAULTLANGUAGE	Language ID for the principal language spoken in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTCOUNTRY	Country code for the principal country in this locale. This is provided so that partially specified locales can be completed with default values.
NLS$LI_IDEFAULTANSICODEPAGE	ANSI code page associated with this locale.
NLS$LI_IDEFAULTOEMCODEPAGE	OEM code page associated with the locale.
NLS$LI_SLIST	Character(s) used to separate list items, for example, comma in many locales.
NLS$LI_IMEASURE	This value is 0 if the metric system (S.I.) is used and 1 for the U.S. system of measurements.
NLS$LI_SDECIMAL	The character(s) used as decimal separator. This is restricted such that it cannot be set to digits 0 - 9.
NLS$LI_STHOUSAND	The character(s) used as separator between groups of digits left of the decimal. This is restricted such that it cannot be set to digits 0 - 9.

Parameter	Description
NLS$LI_SGROUPING	Sizes for each group of digits to the left of the decimal. An explicit size is needed for each group; sizes are separated by semicolons. If the last value is 0 the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_IDIGITS	The number of decimal digits.
NLS$LI_ILZERO	Determines whether to use leading zeros in decimal fields: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_INEGNUMBER	Determines how negative numbers are represented: 0 - Puts negative numbers in parentheses: (1.1) 1 - Puts a minus sign in front: -1.1 2 - Puts a minus sign followed by a space in front: - 1.1 3 - Puts a minus sign after: 1.1- 4 - Puts a space then a minus sign after: 1.1 -
NLS$LI_SNATIVEDIGITS	The ten characters that are the native equivalent to the ASCII 0-9.
NLS$LI_SCURRENCY	The string used as the local monetary symbol. Cannot be set to digits 0-9.

Parameter	Description
NLS$LI_SINTLSYMBOL	Three characters of the International monetary symbol specified in ISO 4217 "Codes for the Representation of Currencies and Funds", followed by the character separating this string from the amount.
NLS$LI_SMONDECIMALSEP	The character(s) used as monetary decimal separator. This is restricted such that it cannot be set to digits 0-9.
NLS$LI_SMONTHOUSANDSEP	The character(s) used as monetary separator between groups of digits left of the decimal. Cannot be set to digits 0-9.
NLS$LI_SMONGROUPING	Sizes for each group of monetary digits to the left of the decimal. If the last value is 0, the preceding value is repeated. To group thousands, specify "3;0".
NLS$LI_ICURRDIGITS	Number of decimal digits for the local monetary format.
NLS$LI_IINTLCURRDIGITS	Number of decimal digits for the international monetary format.
NLS$LI_ICURRENCY	Determines how positive currency is represented: 0 - Puts currency symbol in front with no separation: $1.1 1 - Puts currency symbol in back with no separation: 1.1$ 2 - Puts currency symbol in front with single space after: $ 1.1 3 - Puts currency symbol in back with single space before: 1.1 $

Parameter	Description
NLS$LI_INEGCURR	Determines how negative currency is represented: 0 ($1.1) 1 -$1.1 2 $-1.1 3 $1.1- 4 (1.1$) 5 -1.1$ 6 1.1-$ 7 1.1$- 8 -1.1 $ (space before $) 9 -$ 1.1 (space after $) 10 1.1 $- (space before $) 11 $ 1.1- (space after $) 12 $ -1.1 (space after $) 13 1.1- $ (space before $) 14 ($ 1.1) (space after $) 15 (1.1 $) (space before $)
NLS$LI_SPOSITIVESIGN	String value for the positive sign. Cannot be set to digits 0-9.
NLS$LI_SNEGATIVESIGN	String value for the negative sign. Cannot be set to digits 0-9.
NLS$LI_IPOSSIGNPOSN	Determines the formatting index for positive values: 0 - Parenthesis surround the amount and the monetary symbol

Parameter	Description
	1 - The sign string precedes the amount and the monetary symbol 2 - The sign string follows the amount and the monetary symbol 3 - The sign string immediately precedes the monetary symbol 4 - The sign string immediately follows the monetary symbol
NLS$LI_INEGSIGNPOSN	Determines the formatting index for negative values. Same values as for NLS$LI_IPOSSIGNPOSN.
NLS$LI_IPOSSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a positive amount.
NLS$LI_IPOSSEPBYSPACE	1 if the monetary symbol is separated by a space from a positive amount; otherwise, 0.
NLS$LI_INEGSYMPRECEDES	1 if the monetary symbol precedes, 0 if it follows a negative amount.
NLS$LI_INEGSEPBYSPACE	1 if the monetary symbol is separated by a space from a negative amount; otherwise, 0.
NLS$LI_STIMEFORMAT	Time formatting string. See NLS Date and Time Formatfor explanations of the valid strings.
NLS$LI_STIME	Character(s) for the time separator. Cannot be set to digits 0-9.
NLS$LI_ITIME	Time format: 0 - Use 12-hour format 1 - Use 24-hour format

Parameter	Description
NLS$LI_ITLZERO	Determines whether to use leading zeros in time fields: 0 - Use no leading zeros 1 - Use leading zeros for hours
NLS$LI_S1159	String for the AM designator.
NLS$LI_S2359	String for the PM designator.
NLS$LI_SSHORTDATE	Short Date formatting string for this locale. The d, M and y should have the day, month, and year substituted, respectively. See NLS Date and Time Formatfor explanations of the valid strings.
NLS$LI_SDATE	Character(s) for the date separator. Cannot be set to digits 0-9.
NLS$LI_IDATE	Short Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_ICENTURY	Specifies whether to use full 4-digit century for the short date only: 0 - Two-digit year 1 - Full century
NLS$LI_IDAYLZERO	Specifies whether to use leading zeros in day fields for the short date only: 0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_IMONLZERO	Specifies whether to use leading zeros in month fields for the short date only:

Parameter	Description
	0 - Use no leading zeros 1 - Use leading zeros
NLS$LI_SLONGDATE	Long Date formatting string for this locale. The string returned may contain a string within single quotes (' '). Any characters within single quotes should be left as is. The d, M and y should have the day, month, and year substituted, respectively.
NLS$LI_ILDATE	Long Date format ordering: 0 - Month-Day-Year 1 - Day-Month-Year 2 - Year-Month-Day
NLS$LI_ICALENDARTYPE	Specifies which type of calendar is currently being used: 1 - Gregorian (as in United States) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China 5 - Tangun Era (Korea)
NLS$LI_IOPTIONALCALENDAR	Specifies which additional calendar types are valid and available for this locale. This can be a null separated list of all valid optional calendars: 0 - No additional types valid 1 - Gregorian (localized) 2 - Gregorian (English strings always) 3 - Era: Year of the Emperor (Japan) 4 - Era: Year of the Republic of China

Parameter	Description
	5 - Tangun Era (Korea)
NLS$LI_IFIRSTDAYOFWEEK	Specifies which day is considered first in a week: 0 - SDAYNAME1 1 - SDAYNAME2 2 - SDAYNAME3 3 - SDAYNAME4 4 - SDAYNAME5 5 - SDAYNAME6 6 - SDAYNAME7
NLS$LI_IFIRSTWEEKOFYEAR	Specifies which week of the year is considered first: 0 - Week containing 1/1 1 - First full week following 1/1 2 - First week containing at least 4 days
NLS$LI_SDAYNAME1 - NLS$LI_SDAYNAME7	Native name for each day of the week. 1 = Monday, 2 = Tuesday, etc.
NLS$LI_SABBREVDAYNAME1 - NLS$LI_SABBREVDAYNAME7	Native abbreviated name for each day of the week. 1 = Mon, 2 = Tue, etc.
NLS$LI_SMONTHNAME1 - NLS$LI_SMONTHNAME13	Native name for each month. 1 = January, 2 = February, etc. 13 = the 13th month, if it exists in the locale.
NLS$LI_SABBREVMONTHNAME1 - NLS$LI_SABBREVMONTHNAME13	Native abbreviated name for each month. 1 = Jan, 2 = Feb, etc. 13 = the 13th month, if it exists in the locale.

When NLSGetLocaleInfo (*type*, *outstr*) returns information about the date and time formats of the current locale, the value returned in *outstr* can be interpreted according to the following tables. Any text returned within a date and time string that is enclosed within single quotes should be left in the string in its exact form; that is, do not change the text or the location within the string.

Day

The day can be displayed in one of four formats using the letter "d". The following table shows the four variations:

d	Day of the month as digits without leading zeros for single-digit days
dd	Day of the month as digits with leading zeros for single-digit days
ddd	Day of the week as a three-letter abbreviation (SABBREVDAYNAME)
dddd	Day of the week as its full name (SDAYNAME)

Month

The month can be displayed in one of four formats using the letter "M". The uppercase "M" distinguishes months from minutes. The following table shows the four variations:

M	Month as digits without leading zeros for single-digit months
MM	Month as digits with leading zeros for single-digit months
MMM	Month as a three-letter abbreviation (SABBREVMONTHNAME)
MMMM	Month as its full name (SMONTHNAME)

Year

The year can be displayed in one of three formats using the letter "y". The following table shows the three variations:

y	Year represented by only the last digit
yy	Year represented by only the last two digits
yyyy	Year represented by the full 4 digits

Period/Era

The period/era string is displayed in a single format using the letters "gg".

gg	Period/Era string

Time

The time can be displayed in one of many formats using the letter "h" or "H" to denote hours, the letter "m" to denote minutes, the letter "s" to denote seconds and the letter "t" to denote the time marker. The following table shows the numerous variations of the time format. Lowercase "h" denotes the 12 hour clock and uppercase "H" denotes the 24 hour clock. The lowercase "m" distinguishes minutes from months.

h	Hours without leading zeros for single-digit hours (12 hour clock)
hh	Hours with leading zeros for single-digit hours (12 hour clock)
H	Hours without leading zeros for single-digit hours (24 hour clock)
HH	Hours with leading zeros for single-digit hours (24 hour clock)
m	Minutes without leading zeros for single-digit minutes
mm	Minutes with leading zeros for single-digit minutes
s	Seconds without leading zeros for single-digit seconds

ss	Seconds with leading zeros for single-digit seconds
t	One-character time marker string
tt	Multicharacter time marker string

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFNLS
INTEGER(4) strlen
CHARACTER(40) str
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME1, str)
print *, str    ! prints Monday if language is English
strlen = NLSGetLocaleInfo(NLS$LI_SDAYNAME2, str)
print *, str    ! prints Tuesday if language is English
```

See Also

- M to N
- NLSGetLocale
- NLSFormatDate
- NLSFormatTime
- NLSSetLocale

NLSSetEnvironmentCodepage (W*32, W*64)

NLS Function: *Sets the codepage for the current console. The specified codepage affects the current console program and any other programs launched from the same console. It does not affect other open consoles or any consoles opened later.*

Module

USE IFNLS

Syntax

```
result = NLSSetEnvironmentCodepage (codepage,flags)
```

codepage	(Input) INTEGER(4). Number of the codepage to set as the console codepage.
flags	(Input) INTEGER(4). Must be set to NLS$ConsoleEnvironmentCodepage.

Results

The result type is INTEGER(4). The result is zero if successful. Otherwise, returns one of the following error codes (defined in `IFNLS.F90`):

- NLS$ErrorInvalidCodepage - *codepage* is invalid or not installed on the system
- NLS$ErrorInvalidFlags - *flags* is not valid
- NLS$ErrorNoConsole - There is no console associated with the given application; so operations, with the console codepage are not possible

The *flags* argument must be NLS$ConsoleEnvironmentCodepage; it cannot be NLS$WindowsEnvironmentCodepage. NLSSetEnvironmentCodepage does not affect the Windows* codepage.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- NLSGetEnvironmentCodepage

Building Applications: Locale Setting and Inquiry Routines

NLSSetLocale (W*32, W*64)

NLS Function: *Sets the current language, country, or codepage.*

Module

USE IFNLS

Syntax

```
result = NLSSetLocale (language[,country] [,codepage])
```

language (Input) Character*(*). One of the languages supported by the Windows* OS NLS APIs.

country (Input; optional) Character*(*). If specified, characterizes the language further. If omitted, the default country for the language is set.

codepage (Input; optional) INTEGER(4). If specified, codepage to use for all character-oriented NLS functions. Can be any valid supported codepage or one of the following predefined values (defined in `IFNLS.F90`):

- NLS$CurrentCodepage - The codepage is not changed. Only the language and country settings are altered by the function.
- NLS$ConsoleEnvironmentCodepage - The codepage is changed to the default environment codepage currently in effect for console programs.
- NLS$ConsoleLanguageCodepage - The codepage is changed to the default console codepage for the language and country combination specified.
- NLS$WindowsEnvironmentCodepage - The codepage is changed to the default environment codepage currently in effect for Windows* OS programs.
- NLS$WindowsLanguageCodepage - The codepage is changed to the default Windows OS codepage for the language and country combination specified.

If you omit *codepage*, it defaults to NLS$WindowsLanguageCodepage. At program startup, NLS$WindowsEnvironmentCodepage is used to set the codepage.

Results

The result type is INTEGER(4). The result is zero if successful. Otherwise, one of the following error codes (defined in `IFNLS.F90`) may be returned:

- NLS$ErrorInvalidLanguage - *language* is invalid or not supported
- NLS$ErrorInvalidCountry - *country* is invalid or is not valid with the language specified

- NLS$ErrorInvalidCodepage - *codepage* is invalid or not installed on the system

NOTE. NLSSetLocale works on installed locales only. Many locales are supported, but they must be installed through the system Control Panel/International menu.

When doing mixed-language programming with Fortran and C, calling NLSSetLocale with a codepage other than the default environment Windows OS codepage causes the codepage in the C run-time library to change by calling C's setmbcp( ) routine with the new codepage. Conversely, changing the C run-time library codepage does not change the codepage in the Fortran NLS library.

Calling NLSSetLocale has no effect on the locale used by C programs. The locale set with C's setlocale( ) routine is independent of NLSSetLocale.

Calling NLSSetLocale with the default environment console codepage, NLS$ConsoleEnvironmentCodepage, causes an implicit call to the Windows OS API SetFileApisToOEM( ). Calling NLSSetLocale with any other codepage causes a call to SetFileApisToANSI( ).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- M to N
- NLSGetLocale

FREEFORM and NOFREEFORM

General Compiler Directives: *FREEFORM specifies that source code is in free-form format. NOFREEFORM specifies that source code is in fixed-form format.*

Syntax

```
cDEC$ FREEFORM
cDEC$ NOFREEFORM
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

When the FREEFORM or NOFREEFORM directives are used, they remain in effect for the remainder of the file, or until the opposite directive is used. When in effect, they apply to include files, but do not affect USE modules, which are compiled separately.

See Also

- E to F
- M to N
- Source Forms
- General Compiler Directives
- free compiler option

Building Applications: Compiler Directives Related to Options

OPTIMIZE and NOOPTIMIZE

General Compiler Directive: *Enables or disables optimizations.*

Syntax

```
cDEC$ OPTIMIZE[: n]
cDEC$ NOOPTIMIZE
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is the number denoting the optimization level. The number can be 0, 1, 2, or 3, which corresponds to compiler options O0, O1, O2, and O3. If *n* is omitted, the default is 2, which corresponds to option O2.

The OPTIMIZE and NOOPTIMIZE directives can only appear once at the top of a procedure program unit. A procedure program unit is a main program, an external subroutine or function, or a module. OPTIMIZE and NOOPTIMIZE cannot appear between program units or in a block data program unit. They do not affect any modules invoked with the USE statement in the program unit that contains them. They do affect CONTAINed procedures that do not include an explicit OPTIMIZE or NOOPTIMIZE directive.

NOOPTIMIZE is the same as OPTIMIZE:0. They are both equivalent to –O0 (Linux and Mac OS X) and /Od (Windows).

The procedure is compiled with an optimization level equal to the smaller of *n* and the optimization level specified by the O compiler option on the command line. For example, if the procedure contains the directive NOOPTIMIZE and the program is compiled with the O3 command line option, this procedure is compiled at O0 while the rest of the program is compiled at O3.

See Also

- M to N
- O to P
- General Compiler Directives
- O compiler option

PREFETCH and NOPREFETCH

General Compiler Directives: *PREFETCH enables a data prefetch from memory. Prefetching data can minimize the effects of memory latency. NOPREFETCH (the default) disables data prefetching. These directives affect the heuristics used in the compiler.*

Syntax

```
cDEC$ PREFETCH [var1[: hint1[: distance1]] [,var2[: hint2[: distance2]]]...]
cDEC$ NOPREFETCH [var1[,var2]...]
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
var	Is an optional memory reference.
hint	Is an optional integer initialization expression with the value 0, 1, 2, or 3. These are the same as the values for *hint* in the intrinsic subroutine MM_PREFETCH. To use this argument, you must also specify *var*.
distance	Is an optional integer initialization expression with a value greater than 0. It indicates the number of loop iterations to perform before the prefetch. To use this argument, you must also specify *var* and *hint*.

To use these directives, compiler option O2 or O3 must be set.

This directive affects the DO loop it precedes.

If you specify PREFETCH with no arguments, all arrays accessed in the DO loop will be prefetched.

If a loop includes expression A(j), placing cDEC$ PREFETCH A in front of the loop instructs the compiler to insert prefetches for A(j + d) within the loop. The d is determined by the compiler.

Example

```
cDEC$ NOPREFETCH c
cDEC$ PREFETCH a
do i = 1, m
  b(i) = a(c(i)) + 1
enddo
```

The following example is valid on IA-64 architecture:

```
          sum = 0.d0
          do j=1,lastrow-firstrow+1
             i = rowstr(j)
             iresidue = mod( rowstr(j+1)-i, 8 )
             sum = 0.d0
CDEC$ NOPREFETCH a,p,colidx
             do k=i,i+iresidue-1
                sum = sum +  a(k)*p(colidx(k))
             enddo
CDEC$ NOPREFETCH colidx
CDEC$ PREFETCH a:1:40
CDEC$ PREFETCH p:1:20
             do k=i+iresidue, rowstr(j+1)-8, 8
                sum = sum + a(k  )*p(colidx(k  ))
     &                    + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
     &                    + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
     &                    + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
     &                    + a(k+7)*p(colidx(k+7))
             enddo
          q(j) = sum
          enddo
```

See Also

- M to N
- O to P
- MM_PREFETCH
- O

Optimizing Applications: Prefetching Support

STRICT and NOSTRICT

General Compiler Directive: *STRICT disables language features not found in the language standard specified on the command line (Fortran 2003, Fortran 95, or Fortran 90). NOSTRICT (the default) enables these features.*

Syntax

*c*DEC$ STRICT

*c*DEC$ NOSTRICT

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 2003.

The STRICT and NOSTRICT directives can appear only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. STRICT and NOSTRICT cannot appear between program units, or at the beginning of internal subprograms. They do not affect any modules invoked with the USE statement in the program unit that contains them.

Example

```
! NOSTRICT by default
TYPE stuff
  INTEGER(4) k
  INTEGER(4) m
  CHARACTER(4) name
END TYPE stuff
TYPE (stuff) examp
DOUBLE COMPLEX cd  ! non-standard data type, no error
cd =(3.0D0, 4.0D0)
examp.k = 4        ! non-standard component designation,
                   ! no error
END
SUBROUTINE STRICTDEMO( )
  !DEC$ STRICT
   TYPE stuff
    INTEGER(4) k
    INTEGER(4) m
    CHARACTER(4) name
   END TYPE stuff
   TYPE (stuff) samp
   DOUBLE COMPLEX cd   ! ERROR
   cd =(3.0D0, 4.0D0)
   samp.k = 4          ! ERROR
END SUBROUTINE
```

See Also

- M to N
- S

- General Compiler Directives
- stand compiler option

Building Applications: Compiler Directives Related to Options

SWP and NOSWP (i64 only)

General Compiler Directives: *SWP enables software pipelining for a DO loop. NOSWP (the default) disables this software pipelining. These directives are only available on IA-64 architecture.*

Syntax

*c*DEC$ SWP

*c*DEC$ NOSWP

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The SWP directive must precede the DO statement for each DO loop it affects.

The SWP directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow.

The software pipelining optimization specified by the SWP directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages.

This allows increased instruction level parallelism, which can reduce the impact of long-latency operations, resulting in faster loop execution.

Loops chosen for software pipelining are always innermost loops containing procedure calls that are inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see compiler option -funroll-loops or /Qunroll).

You can request and view the optimization report to see whether software pipelining was applied.

Example

```
!DEC$ SWP
  do i = 1, m
    if (a(i) .eq. 0) then
      b(i) = a(i) + 1
    else
      b(i) = a(i)/c(i)
    endif
  enddo
```

See Also

- M to N
- S
- Syntax Rules for Compiler Directives
- Rules for General Directives that Affect DO Loops
- unroll, Qunroll compiler option

Optimizing Applications: Pipelining for Itanium®-based Architecture

Optimizing Applications: Optimizer Report Generation

NOT

Elemental Intrinsic Function (Generic):
Returns the logical complement of the argument.

Syntax

```
result = NOT (i)
```

i (Input) Must be of type integer.

Results

The result type is the same as *i*. The result value is obtained by complementing *i* bit-by-bit according to the following truth table:

I	NOT (I)
1	0
0	1

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Specific Name	Argument Type	Result Type
BNOT	INTEGER(1)	INTEGER(1)
INOT[1]	INTEGER(2)	INTEGER(2)
JNOT	INTEGER(4)	INTEGER(4)
KNOT	INTEGER(8)	INTEGER(8)

[1] Or HNOT.

Example

If I has a value equal to 10101010 (base 2), NOT (I) has the value 01010101 (base 2).

The following shows another example:

```
INTEGER(2) i(2), j(2)
i = (/4, 132/)          ! i(1) = 0000000000000100
                        ! i(2) = 0000000010000100
j = NOT(i)              ! returns (-5,-133)
                        ! j(1) = 1111111111111011
                        ! j(2) = 1111111101111011
```

See Also

- M to N
- BTEST

- IAND
- IBCHNG
- IBCLR
- IBSET
- IEOR
- IOR
- ISHA
- ISHC
- ISHL
- ISHFT
- ISHFTC

UNROLL and NOUNROLL

General Compiler Directive: *Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop. These directives can only be applied to iterative DO loops.*

Syntax

*c*DEC$ UNROLL [(*n*)] -or- *c*DEC$ UNROLL [=*n*]

*c*DEC$ NOUNROLL

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is an integer constant. The range of *n* is 0 through 255.

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

To use these directives, compiler option O2 or O3 must be set.

Example

```
cDEC$ UNROLL
  do i =1, m
    b(i) = a(i) + 1
    d(i) = c(i) + 1
  enddo
```

See Also

- M to N
- T to Z
- General Compiler Directives
- Rules for General Directives that Affect DO Loops
- O compiler option

VECTOR ALWAYS and NOVECTOR

General Compiler Directive: *Enables or disables vectorization of a DO loop.*

Syntax

```
cDEC$ VECTOR ALWAYS
cDEC$ NOVECTOR
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The VECTOR ALWAYS and NOVECTOR directives override the default behavior of the compiler. The VECTOR ALWAYS directive also overrides efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized. You should use the IVDEP directive to ignore assumed dependences.

CAUTION. The directive VECTOR ALWAYS should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.

Example

The compiler normally does not vectorize DO loops that have a large number of non-unit stride references (compared to the number of unit stride references).

In the following example, vectorization would be disabled by default, but the directive overrides this behavior:

```
!DEC$ VECTOR ALWAYS
  do i = 1, 100, 2
    ! two references with stride 2 follow
    a(i) = b(i)
  enddo
```

There may be cases where you want to explicitly avoid vectorization of a loop; for example, if vectorization would result in a performance regression rather than an improvement. In these cases, you can use the NOVECTOR directive to disable vectorization of the loop.

In the following example, vectorization would be performed by default, but the directive overrides this behavior:

```
!DEC$ NOVECTOR
  do i = 1, 100
    a(i) = b(i) + c(i)
  enddo
```

See Also

- M to N
- T to Z
- Rules for General Directives that Affect DO Loops

NULL

Transformational Intrinsic Function (Generic):
Initializes a pointer as disassociated when it is declared. This is a new intrinsic function in Fortran 95.

Syntax

```
result = NULL ([ mold])
```

mold (Optional) Must be a pointer; it can be of any type. Its pointer association status can be associated, disassociated, or undefined. If its status is associated, the target does not have to be defined with a value.

Results

The result type is the same as *mold*, if present; otherwise, it is determined as follows:

If NULL ( ) Appears...	Type is Determined From...
On the right side of pointer assignment	The pointer on the left side
As initialization for an object in a declaration	The object
As default initialization for a component	The component
In a structure constructor	The corresponding component
As an actual argument	The corresponding dummy argument
In a DATA statement	The corresponding pointer object

The result is a pointer with disassociated association status.

CAUTION. If you use module IFWIN or IFWINTY, you will have a name conflict if you use the NULL intrinsic. To avoid this problem, rename the integer parameter constant NULL to something else; for example:

```
USE IFWIN, NULL0 => NULL
```

This example lets you use both NULL0 and NULL( ) in the same program unit with no conflict.

Example

Consider the following:

```
INTEGER, POINTER :: POINT1 => NULL( )
```

This statement defines the initial association status of POINT1 to be disassociated.

NULLIFY

Statement: *Disassociates a pointer from a target.*

Syntax

NULLIFY (*pointer-object*[,*pointer-object*]...)

pointer-object	Is a structure component or the name of a variable; it must be a pointer (have the POINTER attribute).

Description

The initial association status of a pointer is undefined. You can use NULLIFY to initialize an undefined pointer, giving it disassociated status. Then the pointer can be tested using the intrinsic function ASSOCIATED.

Example

The following is an example of the NULLIFY statement:

```
REAL, TARGET  :: TAR(0:50)
REAL, POINTER :: PTR_A(:), PTR_B(:)
PTR_A => TAR
PTR_B => TAR
...
NULLIFY(PTR_A)
```

After these statements are executed, PTR_A will have disassociated status, while PTR_B will continue to be associated with variable TAR.

The following shows another example:

```
!    POINTER2.F90    Pointing at a Pointer and Target
!DEC$ FIXEDFORMLINESIZE:80
     REAL, POINTER :: arrow1 (:)
     REAL, POINTER :: arrow2 (:)
     REAL, ALLOCATABLE, TARGET :: bullseye (:)
     ALLOCATE (bullseye (7))
     bullseye = 1.
     bullseye (1:7:2) = 10.
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye
     arrow1 => bullseye
     WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1
     arrow2 => arrow1
     IF (ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is pointed.'
     WRITE (*,'(1x,a,7f8.0)') 'pointer',arrow2
     NULLIFY (arrow2)
     IF (.NOT.ASSOCIATED(arrow2)) WRITE (*,'(/a/)') ' ARROW2 is not pointed.'
     WRITE (*,'( 1x,a,7f8.0)') 'pointer',arrow1
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye
     END
```

See Also

- M to N
- ALLOCATE
- ASSOCIATED
- DEALLOCATE
- POINTER
- TARGET
- NULL
- Pointer Assignments
- Dynamic Allocation

O to P

OBJCOMMENT

General Compiler Directive: *Specifies a library search path in an object file.*

Syntax

*c*DEC$ OBJCOMMENT LIB: *library*

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
library	Is a character constant specifying the name and, if necessary, the path of the library that the linker is to search.

The linker searches for the library named in OBJCOMMENT as if you named it on the command line, that is, before default library searches. You can place multiple library search directives in the same source file. Each search directive appears in the object file in the order it is encountered in the source file.

If the OBJCOMMENT directive appears in the scope of a module, any program unit that uses the module also contains the directive, just as if the OBJCOMMENT directive appeared in the source file using the module.

If you want to have the OBJCOMMENT directive in a module, but do not want it in the program units that use the module, place the directive outside the module that is used.

Example

```
!  MOD1.F90
MODULE a
   !DEC$ OBJCOMMENT LIB: "opengl32.lib"
END MODULE a
!  MOD2.F90
!DEC$ OBJCOMMENT LIB: "graftools.lib"
MODULE b
   !
END MODULE b
!  USER.F90
PROGRAM go
   USE a    ! library search contained in MODULE a
            !   included here
   USE b    ! library search not included
END
```

See Also

- O to P
- General Compiler Directives

Building Applications: Compiler Directives Related to Options

OPEN

Statement: *Connects an external file to a unit, creates a new file and connects it to a unit, creates a preconnected file, or changes certain properties of a connection.*

Syntax

OPEN ([UNIT=] *io-unit*[, FILE= *name*] [, ERR= *label*] [, IOSTAT=*i-var*],*slist*)

io-unit Is an external unit specifier.

name Is a character or numeric expression specifying the name of the file to be connected. For more information, see FILE Specifier and STATUS Specifier.

label Is the label of the branch target statement that receives control if an error occurs. For more information, see Branch Specifiers.

i-var Is a scalar integer variable that is defined as a positive integer (the number of the error message) if an error occurs, a negative integer if an end-of-file record is encountered, and zero if no error occurs. For more information, see I/O Status Specifier.

slist Is one or more of the following OPEN specifiers in the form *specifier*= *value* or *specifier* (each specifier can appear only once):

ACCESS	CARRIAGECONTROL	MAXREC	RECORDSIZE
ACTION	CONVERT	MODE	RECORDTYPE
ASSOCIATEVARIABLE	DEFAULTFILE	NAME	SHARE
ASYNCHRONOUS	DELIM	ORGANIZATION	SHARED
BLANK	DISPOSE	PAD	STATUS
BLOCKSIZE	FILE	POSITION	TITLE
BUFFERCOUNT	FORM	READONLY	TYPE
BUFFERED	IOFOCUS	RECL	USEROPEN

The OPEN specifiers and their acceptable values are summarized in the OPEN Statement in the *Language Reference*.
The control specifiers that can be specified in an OPEN statement are discussed in I/O Control List in the *Language Reference*.

Description

The control specifiers ([UNIT=] *io-unit*, ERR= *label*, and IOSTAT= *i-var*) and OPEN specifiers can appear anywhere within the parentheses following OPEN. However, if the UNIT specifier is omitted, the *io-unit* must appear first in the list.

Specifier values that are scalar numeric expressions can be any integer or real expression. The value of the expression is converted to integer data type before it is used in the OPEN statement.

Only one unit at a time can be connected to a file, but multiple OPENs can be performed on the same unit. If an OPEN statement is executed for a unit that already exists, the following occurs:

- If FILE is not specified, or FILE specifies the same file name that appeared in a previous OPEN statement, the current file remains connected.

  If the file names are the same, the values for the BLANK, CARRIAGECONTROL, CONVERT, DELIM, DISPOSE, ERR, IOSTAT, and PAD specifiers can be changed. Other OPEN specifier values cannot be changed, and the file position is unaffected.
- If FILE specifies a different file name, the previous file is closed and the new file is connected to the unit.

The ERR and IOSTAT specifiers from any previously executed OPEN statement have no effect on any currently executing OPEN statement. If an error occurs, no file is opened or created.

Secondary operating system messages do not display when IOSTAT is specified. To display these messages, remove IOSTAT or use a platform-specific method.

Example

You can specify character values at run time by substituting a character expression for a specifier value in the OPEN statement. The character value can contain trailing blanks but not leading or embedded blanks; for example:

```
CHARACTER*6 FINAL /' '/
...
IF (expr) FINAL = 'DELETE'
OPEN (UNIT=1, STATUS='NEW', DISP=FINAL)
```

The following statement creates a new sequential formatted file on unit 1 with the default file name fort.1:

```
OPEN (UNIT=1, STATUS='NEW', ERR=100)
```

The following statement creates a file on magnetic tape:

```
OPEN (UNIT=I, FILE='/dev/rmt8',                                  &
     STATUS='NEW', ERR=14, RECL=1024)
```

The following statement opens the file (created in the previous example) for input:

```
OPEN (UNIT=I, FILE='/dev/rmt8', READONLY, STATUS='OLD',          &
     RECL=1024)
```

The following example opens the existing file /usr/users/someone/test.dat:

```
   OPEN (unit=10, DEFAULTFILE='/usr/users/someone/', FILE='test.dat',
  1     FORM='FORMATTED', STATUS='OLD')
```

The following example opens a new file:

```
! Prompt user for a filename and read it:
CHARACTER*64 filename
WRITE (*, '(A\)') ' enter file to create: '
READ (*, '(A)') filename
! Open the file for formatted sequential access as unit 7.
! Note that the specified access need not have been specified,
! since it is the default (as is "formatted").
OPEN (7, FILE = filename, ACCESS = 'SEQUENTIAL', STATUS = 'NEW')
The following example opens an existing file called DATA3.TXT:
! Open a file created by an editor, "DATA3.TXT", as unit 3:
OPEN (3, FILE = 'DATA3.TXT')
```

See Also

- O to P
- READ
- WRITE
- CLOSE
- FORMAT
- INQUIRE
- OPEN Statement

OPTIONAL

Statement and Attribute: *Permits dummy arguments to be omitted in a procedure reference.*

Syntax

The OPTIONAL attribute can be specified in a type declaration statement or an OPTIONAL statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,]  OPTIONAL [, att-ls] :: d-arg[, d-arg]...
```

Statement:

```
OPTIONAL [::] d-arg[, d-arg] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
d-arg	Is the name of a dummy argument.

Description

The OPTIONAL attribute can only appear in the scoping unit of a subprogram or an interface body, and can only be specified for dummy arguments. It cannot be specified for arguments that are passed by value.

A dummy argument is "present" if it associated with an actual argument. A dummy argument that is not optional must be present. You can use the PRESENT intrinsic function to determine whether an optional dummy argument is associated with an actual argument.

To call a procedure that has an optional argument, you must use an explicit interface.

If argument keywords are not used, argument association is positional. The first dummy argument becomes associated with the first actual argument, and so on. If argument keywords are used, arguments are associated by the keyword name, so actual arguments can be in a different order than dummy arguments. A keyword is required for an argument only if a preceding optional argument is omitted or if the argument sequence is changed.

Example

The following example shows a type declaration statement specifying the OPTIONAL attribute:

```
SUBROUTINE TEST(A)

REAL, OPTIONAL, DIMENSION(-10:2) :: A

END SUBROUTINE
```

The following is an example of the OPTIONAL statement:

```
SUBROUTINE TEST(A, B, L, X)
  OPTIONAL :: B
  INTEGER A, B, L, X
  IF (PRESENT(B)) THEN        ! Printing of B is conditional
    PRINT *, A, B, L, X       !   on its presence
  ELSE
    PRINT *, A, L, X
  ENDIF
END SUBROUTINE
INTERFACE
  SUBROUTINE TEST(ONE, TWO, THREE, FOUR)
    INTEGER ONE, TWO, THREE, FOUR
    OPTIONAL :: TWO
  END SUBROUTINE
END INTERFACE
INTEGER I, J, K, L
I = 1
J = 2
K = 3
L = 4
CALL TEST(I, J, K, L)            ! Prints:  1  2  3  4
CALL TEST(I, THREE=K, FOUR=L)    ! Prints:  1  3  4
END
```

Note that in the second call to subroutine TEST, the second positional (optional) argument is omitted. In this case, all following arguments must be keyword arguments.

The following shows another example:

```
SUBROUTINE ADD (a,b,c,d)
  REAL              a, b, d
  REAL, OPTIONAL :: c
  IF (PRESENT(c)) THEN
    d = a + b + c + d
  ELSE
    d = a + b + d
  END IF
END SUBROUTINE
```

Consider the following:

```
SUBROUTINE EX (a, b, c)
REAL, OPTIONAL :: b,c
```

This subroutine can be called with any of the following statements:

```
CALL EX (x, y, z)   !All 3 arguments are passed.
CALL EX (x)         !Only the first argument is passed.
CALL EX (x, c=z)    !The first optional argument is omitted.
```

Note that you *cannot* use a series of commas to indicate omitted optional arguments, as in the following example:

```
CALL EX (x,,z)   !Invalid statement.
```

This results in a compile-time error.

See Also

- O to P
- PRESENT
- Argument Keywords in Intrinsic Procedures
- Optional Arguments
- Argument Association
- Type Declarations
- Compatible attributes

OPTIONS Statement

Statement: *Overrides or confirms the compiler options in effect for a program unit.*

Syntax

`OPTIONS option[option...]`

option	Is one of the following:	
	/ASSUME =	[NO]UNDERSCORE
	/CHECK =	ALL
		[NO]BOUNDS
		NONE
	/NOCHECK	
	/CONVERT =	BIG_ENDIAN
		CRAY
		FDX
		FGX
		IBM
		LITTLE_ENDIAN
		NATIVE
		VAXD
		VAXG
	/[NO]EXTEND_SOURCE	
	/[NO]F77	
	/[NO]I4	
	/[NO]RECURSIVE	

Note that an option must always be preceded by a slash (/).

Some OPTIONS statement options are equivalent to compiler options.

The OPTIONS statement must be the first statement in a program unit, preceding the PROGRAM, SUBROUTINE, FUNCTION, MODULE, and BLOCK DATA statements.

OPTIONS statement options override compiler options, but only until the end of the program unit for which they are defined. If you want to override compiler options in another program unit, you must specify the OPTIONS statement before that program unit.

Example

The following are valid OPTIONS statements:

```
OPTIONS /CHECK=ALL/F77

OPTIONS /I4
```

See Also

- O to P

For details on compiler options, see your *Compiler Options* reference

Building Applications: OPEN Statement CONVERT Method

OPTIMIZE and NOOPTIMIZE

General Compiler Directive: *Enables or disables optimizations.*

Syntax

```
cDEC$ OPTIMIZE[: n]

cDEC$ NOOPTIMIZE
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is the number denoting the optimization level. The number can be 0, 1, 2, or 3, which corresponds to compiler options O0, O1, O2, and O3. If *n* is omitted, the default is 2, which corresponds to option O2.

The OPTIMIZE and NOOPTIMIZE directives can only appear once at the top of a procedure program unit. A procedure program unit is a main program, an external subroutine or function, or a module. OPTIMIZE and NOOPTIMIZE cannot appear between program units or in a block

data program unit. They do not affect any modules invoked with the USE statement in the program unit that contains them. They do affect CONTAINed procedures that do not include an explicit OPTIMIZE or NOOPTIMIZE directive.

NOOPTIMIZE is the same as OPTIMIZE:0. They are both equivalent to –O0 (Linux and Mac OS X) and /Od (Windows).

The procedure is compiled with an optimization level equal to the smaller of *n* and the optimization level specified by the O compiler option on the command line. For example, if the procedure contains the directive NOOPTIMIZE and the program is compiled with the O3 command line option, this procedure is compiled at O0 while the rest of the program is compiled at O3.

See Also

- M to N
- O to P
- General Compiler Directives
- O compiler option

OPTIONS Directive

General Compiler Directive: *Affects data alignment and warnings about data alignment.*

Syntax

*c*DEC$ OPTIONS *option*[*option*]

...

*c*DEC$ END OPTIONS

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
option	Is one (or both) of the following:
/WARN=[NO]ALIGNMENT	Controls whether warnings are issued by the compiler for data that is not naturally aligned. By default, you receive compiler messages when misaligned data is encountered (/WARN=ALIGNMENT).
/[NO]ALIGN[= *p*]	Controls alignment of fields in record structures and data items in common blocks. The fields and data items can be naturally

aligned (for performance reasons) or they can be packed together on arbitrary byte boundaries.

p Is a specifier with one of the following forms:

`[class=]rule`

`(class= rule,...)`

`ALL`

`NONE`

class Is one of the following keywords:

- `COMMONS`: For common blocks
- `RECORDS`: For records
- `STRUCTURES`: A synonym for RECORDS

rule Is one of the following keywords:

`PACKED` Packs fields in records or data items in common blocks on arbitrary byte boundaries.

`NATURAL` Naturally aligns fields in records and data items in common blocks on up to

64-bit boundaries (inconsistent with the Fortran 95/90 standard). This keyword causes the compiler to naturally align all data in a common block, including INTEGER(KIND=8), REAL(KIND=8), and all COMPLEX data.

STANDARD Naturally aligns data items in common blocks on up to 32-bit boundaries (consistent with the Fortran 95/90 standard). This keyword only applies to common blocks; so, you can specify /ALIGN:COMMONS:STANDARD, but you cannot specify /ALIGN:STANDARD.

ALL	Is the same as specifying OPTIONS /ALIGN, OPTIONS /ALIGN=NATURAL, and OPTIONS /ALIGN=(RECORDS=NATURAL,COMMONS=NATURAL)
NONE	Is the same as specifying OPTIONS /NOALIGN, OPTIONS /ALIGN=PACKED, and OPTIONS /ALIGN=(RECORDS=PACKED,COMMONS=PACKED)

The OPTIONS (and accompanying END OPTIONS) directives must come after OPTIONS, SUBROUTINE, FUNCTION, and BLOCK DATA statements (if any) in the program unit, and before the executable part of the program unit.

The OPTIONS directive supersedes compiler option align.

For performance reasons, Intel Fortran aligns local data items on natural boundaries. However, EQUIVALENCE, COMMON, RECORD, and STRUCTURE data declaration statements can force misaligned data. If /WARN=NOALIGNMENT is specified, warnings will not be issued if misaligned data is encountered.

NOTE. Misaligned data significantly increases the time it takes to execute a program. As the number of misaligned fields encountered increases, so does the time needed to complete program execution. Specifying /ALIGN (or compiler option align) minimizes misaligned data.

If you want aligned data in common blocks, do one of the following:

- Specify OPTIONS /ALIGN=COMMONS=STANDARD for data items up to 32 bits in length.
- Specify OPTIONS /ALIGN=COMMONS=NATURAL for data items up to 64 bits in length.
- Place source data declarations within the common block in descending size order, so that each data item is naturally aligned.

If you want packed, unaligned data in a record structure, do one of the following:

- Specify OPTIONS /ALIGN=RECORDS=PACKED.

- Place source data declarations in the record structure so that the data is naturally aligned.

Example

```
! directives can be nested up to 100 levels
  CDEC$ OPTIONS /ALIGN=PACKED     ! Start of Group A
    declarations
  CDEC$ OPTIONS /ALIGN=RECO=NATU     ! Start of nested Group B
    more declarations
  CDEC$ END OPTIONS                  ! End of Group B
    still more declarations
  CDEC$ END OPTIONS               ! End of Group A
```

The OPTIONS specification for Group B only applies to RECORDS; common blocks within Group B will be PACKED. This is because COMMONS retains the previous setting (in this case, from the Group A specification).

See Also

- O to P
- General Compiler Directives
- align compiler option

OR

Elemental Intrinsic Function (Generic):
Performs a bitwise inclusive OR on its arguments.
See IOR.

Example

```
INTEGER i
i = OR(3, 10)   ! returns 11
```

ORDERED

OpenMP* Fortran Compiler Directive: *Specifies a block of code to be executed in the order in which iterations would be executed in sequential execution.*

Syntax

```
c$OMP ORDERED
   block
c$OMP ORDERED
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. The DO directive to which the ordered section binds *must* have the ORDERED clause specified.

An iteration of a loop using a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

One thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it can be guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel.

Ordered sections that bind to different DO directives are independent of each other.

Example

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
c$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO
      ...
      SUBROUTINE WORK(K)
c$OMP ORDERED
      WRITE(*,*) K
c$OMP END ORDERED
```

See Also

- O to P
- OpenMP Fortran Compiler Directives

OUTGTEXT (W*32, W*64)

Graphics Subroutine: *In graphics mode, sends a string of text to the screen, including any trailing blanks.*

Module

USE IFQWIN

Syntax

`CALL OUTGTEXT (text)`

text (Input) Character*(*). String to be displayed.

Text output begins at the current graphics position, using the current font set with SETFONT and the current color set with SETCOLORRGB or SETCOLOR. No formatting is provided. After it outputs the text, OUTGTEXT updates the current graphics position.

Before you call OUTGTEXT, you must call the INITIALIZEFONTS function.

Because OUTGTEXT is a graphics function, the color of text is affected by the SETCOLORRGB function, not by SETTEXTCOLORRGB.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! build as a QuickWin App.
USE IFQWIN
INTEGER(2) result
INTEGER(4) i
TYPE (xycoord) xys
result = INITIALIZEFONTS()
result = SETFONT('t''Arial''h18w10pvib')
do i=1,6
   CALL MOVETO(INT2(0),INT2(30*(i-1)),xys)
   grstat=SETCOLOR(INT2(i))
   CALL OUTGTEXT('This should be ')
   SELECT CASE (i)
     CASE (1)
       CALL OUTGTEXT('Blue')
     CASE (2)
       CALL OUTGTEXT('Green')
     CASE (3)
       CALL OUTGTEXT('Cyan')
     CASE (4)
       CALL OUTGTEXT('Red')
     CASE (5)
       CALL OUTGTEXT('Magenta')
     CASE (6)
       CALL OUTGTEXT('Orange')
   END SELECT
end do
END
```

See Also

- O to P
- GETFONTINFO
- GETGTEXTEXTENT
- INITIALIZEFONTS
- MOVETO
- SETCOLORRGB
- SETFONT
- SETGTEXTROTATION

Building Applications: Setting Figure Properties

Building Applications: Selecting Display Options

OUTTEXT (W*32, W*64)

Graphics Subroutine: *In text or graphics mode, sends a string of text to the screen, including any trailing blanks.*

Module

USE IFQWIN

Syntax

`CALL OUTTEXT (text)`

text (Input) Character*(*). String to be displayed.

Text output begins at the current text position in the color set with SETTEXTCOLORRGB or SETTEXTCOLOR. No formatting is provided. After it outputs the text, OUTTEXT updates the current text position.

To output text using special fonts, you must use the OUTGTEXT subroutine.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) oldcolor
TYPE (rccoord) rc
CALL CLEARSCREEN($GCLEARSCREEN)
CALL SETTEXTPOSITION (INT2(1), INT2(5), rc)
oldcolor = SETTEXTCOLOR(INT2(4))
CALL OUTTEXT ('Hello, everyone')
END
```

See Also

- O to P
- OUTGTEXT
- SETTEXTPOSITION
- SETTEXTCOLORRGB
- WRITE
- WRAPON

Building Applications: Displaying Character-Based Text

Building Applications: Using Text Colors

PACK Function

Transformational Intrinsic Function (Generic):
Takes elements from an array and packs them into a rank-one array under the control of a mask.

Syntax

`result = PACK (array,mask[,vector])`

array	(Input) Must be an array. It may be of any data type.
mask	(Input) Must be of type logical and conformable with *array*. It determines which elements are taken from *array*.

vector	(Input; optional) Must be a rank-one array with the same type and type parameters as *array*. Its size must be at least *t*, where *t* is the number of true elements in *mask*. If *mask* is a scalar with value true, *vector* must have at least as many elements as there are in array. Elements in *vector* are used to fill out the result array if there are not enough elements selected by *mask*.

Results

The result is a rank-one array with the same type and type parameters as *array*. If *vector* is present, the size of the result is that of *vector*. Otherwise, the size of the result is the number of true elements in *mask*, or the number of elements in *array* (if *mask* is a scalar with value true).

Elements in *array* are processed in array element order to form the result array. Element i of the result is the element of *array* that corresponds to the ith true element of *mask*. If *vector* is present and has more elements than there are true values in *mask*, any result elements that are empty (because they were not true according to *mask*) are set to the corresponding values in *vector*.

Example

N is the array

```
[ 0  8  0 ]
[ 0  0  0 ]
[ 7  0  0 ].
```

PACK (N, MASK=N .NE. 0, VECTOR=(/1, 3, 5, 9, 11, 13/)) produces the result (7, 8, 5, 9, 11, 13).

PACK (N, MASK=N .NE. 0) produces the result (7, 8).

The following shows another example:

```
INTEGER array(2, 3), vec1(2), vec2(5)
LOGICAL mask (2, 3)
array = RESHAPE((/7, 0, 0, -5, 0, 0/), (/2, 3/))
mask = array .NE. 0
!  array is  7  0 0  and mask is  T F F
!            0 -5 0               F T F
VEC1 = PACK(array, mask)      ! returns ( 7, -5 )
VEC2 = PACK(array, array .GT. 0, VECTOR= (/1,2,3,4,5/))
!  returns ( 7, 2, 3, 4, 5 )
```

See Also

- O to P
- UNPACK

PACK Directive

General Compiler Directive: *Specifies the memory starting addresses of derived-type items (and record structure items).*

Syntax

*c*DEC$ PACK[: [{1 | 2 | 4 | 8}]]

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

Items of derived types, unions, and structures are aligned in memory on the smaller of two sizes: the size of the type of the item, or the current alignment setting. The current alignment setting can be 1, 2, 4, or 8 bytes. The default initial setting is 8 bytes (unless compiler option vms or align rec *n* bytes is specified). By reducing the alignment setting, you can pack variables closer together in memory.

The PACK directive lets you control the packing of derived-type or record structure items inside your program by overriding the current memory alignment setting.

For example, if PACK:1 is specified, all variables begin at the next available byte, whether odd or even. Although this slightly increases access time, no memory space is wasted. If PACK:4 is specified, INTEGER(1), LOGICAL(1), and all character variables begin at the next available byte, whether odd or even. INTEGER(2) and LOGICAL(2) begin on the next even byte; all other variables begin on 4-byte boundaries.

If the PACK directive is specified without a number, packing reverts to the compiler option setting (if any), or the default setting of 8.

The directive can appear anywhere in a program before the derived-type definition or record structure definition. It cannot appear *inside* a derived-type or record structure definition.

Example

```
! Use 4-byte packing for this derived type
! Note PACK is used outside of the derived type definition
!DEC$ PACK:4
TYPE pair
   INTEGER a, b
END TYPE
! revert to default or compiler option
!DEC$ PACK
```

See Also

- O to P
- TYPE
- STRUCTURE...END STRUCTURE
- UNION...END UNION
- General Compiler Directives
- align rec *n* bytes compiler option
- vms compiler option

Building Applications: Compiler Directives Related to Options

PACKTIMEQQ

Portability Subroutine: *Packs time and date values.*

Module

USE IFPORT

Syntax

CALL PACKTIMEQQ (*timedate*,*iyr*,*imon*,*iday*,*ihr*,*imin*,*isec*)

timedate	(Output) INTEGER(4). Packed time and date information.
iyr	(Input) INTEGER(2). Year (*xxxx*AD).
imon	(Input) INTEGER(2). Month (1 - 12).
iday	(Input) INTEGER(2). Day (1 - 31)
ihr	(Input) INTEGER(2). Hour (0 - 23)
imin	(Input) INTEGER(2). Minute (0 - 59)
isec	(Input) INTEGER(2). Second (0 - 59)

The packed time is the number of seconds since 00:00:00 Greenwich mean time, January 1, 1970. Because packed time values can be numerically compared, you can use PACKTIMEQQ to work with relative date and time values. Use UNPACKTIMEQQ to unpack time information. SETFILETIMEQQ uses packed time.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS DLL LIB

Example

```
USE IFPORT
INTEGER(2) year, month, day, hour, minute, second,   &
           hund
INTEGER(4) timedate
CALL GETDAT (year, month, day)
CALL GETTIM (hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day, hour,   &
                 minute, second)
END
```

See Also

- O to P
- UNPACKTIMEQQ
- SETFILETIMEQQ
- GETFILEINFOQQ
- TIME portability routine

PARALLEL Directive (OpenMP*)

OpenMP* Fortran Compiler Directive: *Defines a parallel region.*

Syntax

```
c$OMP PARALLEL [clause[[,] clause] ... ]
   block
c$OMP END PARALLEL
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Is one of the following: • COPYIN (list) • DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE) • FIRSTPRIVATE (list)

- IF (*scalar_logical_expression*)

  Specifies that the enclosed code section is to be executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. If this clause is not used, the region is executed as if an IF(.TRUE.) clause were specified.

  This clause is evaluated by the master thread before any data scope attributes take effect.

  Only a single IF clause can appear in the directive.

- NUM_THREADS (*scalar_integer_expression*)

  Specifies the number of threads to be used in a parallel region. The *scalar_integer_expression* must evaluate to a positive scalar integer value. Only a single NUM_THREADS clause can appear in the directive.

- PRIVATE (list)
- REDUCTION (operator | intrinsic : list)
- SHARED (list)

block Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

The PARALLEL and END PARALLEL directive pair must appear in the same routine in the executable section of the code.

The END PARALLEL directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution at the end of a parallel region.

The number of threads in the team can be controlled by the NUM_THREADS clause, the environment variable OMP_NUM_THREADS, or by calling the run-time library routine OMP_SET_NUM_THREADS from a serial portion of the program.

NUM_THREADS supersedes the OMP_SET_NUM_THREADS routine, which supersedes the OMP_NUM_THREADS environment variable. Subsequent parallel regions, however, are not affected unless they have their own NUM_THREADS clauses.

Once specified, the number of threads in the team remains constant for the duration of that parallel region.

If the dynamic threads mechanism is enabled by an environment variable or a library routine, then the number of threads requested by the NUM_THREADS clause is the maximum number to use in the parallel region.

The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads.

If a thread executing a parallel region encounters another parallel region, it creates a new team and becomes the master of that new team. By default, nested parallel regions are always serialized and executed by a team of one thread.

Example

You can use the PARALLEL directive in coarse-grain parallel programs. In the following example, each thread in the parallel region decides what part of the global array X upon which to work based on the thread number:

```
c$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM( )
      NP = OMP_GET_NUM_THREADS( )
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
c$OMP END PARALLEL
```

Assuming you previously used the environment variable OMP_NUM_THREADS to set the number of threads to six, you can change the number of threads between parallel regions as follows:

```
      CALL OMP_SET_NUM_THREADS(3)
!$OMP PARALLEL
...
!$OMP END PARALLEL
      CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
...
!$OMP END PARALLEL DO
```

For more information on environment variables, see *Building Applications*.

See Also

- O to P
- OpenMP Fortran Compiler Directives
- OpenMP* Fortran Routines
- PARALLEL DO
- SHARED Clause

PARALLEL and NOPARALLEL Loop Directives

General Compiler Directives: *PARALLEL facilitates auto-parallelization for the immediately following DO loop. NOPARALLEL prevents this auto-parallelization.*

Syntax

```
cDEC$ PARALLEL [ALWAYS]
cDEC$ NOPARALLEL
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

PARALLEL helps the compiler to resolve dependencies, facilitating auto-parallelization of the immediately following DO loop. It instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the loop. However, if dependencies are proven, they are not ignored.

In addition, PARALLEL ALWAYS overrides the compiler heuristics that estimate the likelihood that parallelization of a loop will increase performance. It allows a loop to be parallelized even if the compiler thinks parallelization may not improve performance.

NOPARALLEL prevents auto-parallelization of the immediately following DO loop.

These directives take effect only if you specify the compiler option that enables auto-parallelization.

CAUTION. The directive PARALLEL ALWAYS should be used with care. Overriding the heuristics of the compiler should only be done if you are absolutely sure the parallelization will improve performance.

Example

```
program main
parameter (n=100)
integer x(n),a(n)
!DEC$ NOPARALLEL
   do i=1,n
     x(i) = i
   enddo
!DEC$ PARALLEL
   do i=1,n
     a( x(i) ) = i
   enddo
end
```

See Also

- O to P
- O to P
- Rules for General Directives that Affect DO Loops

PARALLEL and NOPARALLEL Loop Directives

General Compiler Directives: *PARALLEL facilitates auto-parallelization for the immediately following DO loop. NOPARALLEL prevents this auto-parallelization.*

Syntax

*c*DEC$ PARALLEL [ALWAYS]

*c*DEC$ NOPARALLEL

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

PARALLEL helps the compiler to resolve dependencies, facilitating auto-parallelization of the immediately following DO loop. It instructs the compiler to ignore dependencies that it assumes may exist and which would prevent correct parallelization in the loop. However, if dependencies are proven, they are not ignored.

In addition, PARALLEL ALWAYS overrides the compiler heuristics that estimate the likelihood that parallelization of a loop will increase performance. It allows a loop to be parallelized even if the compiler thinks parallelization may not improve performance.

NOPARALLEL prevents auto-parallelization of the immediately following DO loop.

These directives take effect only if you specify the compiler option that enables auto-parallelization.

CAUTION. The directive PARALLEL ALWAYS should be used with care. Overriding the heuristics of the compiler should only be done if you are absolutely sure the parallelization will improve performance.

Example

```
program main
parameter (n=100)
integer x(n),a(n)
!DEC$ NOPARALLEL
   do i=1,n
     x(i) = i
   enddo
!DEC$ PARALLEL
   do i=1,n
     a( x(i) ) = i
   enddo
end
```

See Also

- O to P
- O to P

- Rules for General Directives that Affect DO Loops

PARALLEL DO

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single DO directive.*

Syntax

```
c$OMP PARALLEL DO [clause[[,] clause] ... ]
      do-loop
[c$OMP END PARALLEL DO]
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Can be any of the clauses accepted by the DO or PARALLEL directives.
do-loop	Is a DO iteration (a DO loop). It cannot be a DO WHILE or a DO loop without loop control. The DO loop iteration variable must be of type integer. You cannot branch out of a DO loop associated with a DO directive.

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

Example

In the following example, the loop iteration variable is private by default and it is not necessary to explicitly declare it. The END PARALLEL DO directive is optional:

```
c$OMP PARALLEL DO
      DO I=1,N
          B(I) = (A(I) + A(I-1)) / 2.0
      END DO
c$OMP END PARALLEL DO
```

The following example shows how to use the REDUCTION clause in a PARALLEL DO directive:

```
c$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      END DO
c$OMP END PARALLEL DO
```

See Also

- O to P
- OpenMP Fortran Compiler Directives

PARALLEL SECTIONS

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.*

Syntax

```
c$OMP PARALLEL SECTIONS [clause[[,] clause] ...]
[c$OMP SECTION]
   block
[c$OMP SECTION
   block]...
c$OMP END PARALLEL SECTIONS
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Can be any of the clauses accepted by the PARALLEL or SECTIONS directives.

block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.

The last section ends at the END PARALLEL SECTIONS directive.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
c$OMP PARALLEL SECTIONS
c$OMP SECTION
      CALL XAXIS
c$OMP SECTION
      CALL YAXIS
c$OMP SECTION
      CALL ZAXIS
c$OMP END PARALLEL SECTIONS
```

See Also

- O to P
- OpenMP Fortran Compiler Directives

PARALLEL WORKSHARE

OpenMP* Fortran Compiler Directive: *Provides an abbreviated way to specify a parallel region containing a single WORKSHARE directive.*

Syntax

```
c$OMP PARALLEL WORKSHARE [clause[[,] clause] ... ]
   block
c$OMP END PARALLEL WORKSHARE
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Is any of the clauses accepted by the PARALLEL or WORKSHARE directives.

block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

See Also

- O to P
- OpenMP Fortran Compiler Directives
- WORKSHARE
- PARALLEL

PARAMETER

Statement and Attribute: *Defines a named constant.*

Syntax

The PARAMETER attribute can be specified in a type declaration statement or a PARAMETER statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,]  PARAMETER [, att-ls] :: c =expr[, c = expr] ...
```

Statement:

```
PARAMETER [( ]c= expr[, c= expr] ... [) ]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
c	Is the name of the constant.
expr	Is an initialization expression. It can be of any data type.

Description

The type, type parameters, and shape of the named constant are determined in one of the following ways:

- By an explicit type declaration statement in the same scoping unit.
- By the implicit typing rules in effect for the scoping unit. If the named constant is implicitly typed, it can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

For example, consider the following statement:

```
PARAMETER (MU=1.23)
```

According to implicit typing, MU is of integer type, so MU=1. For MU to equal 1.23, it should previously be declared REAL in a type declaration or be declared in an IMPLICIT statement.

A named constant must not appear in a format specification or as the character count for Hollerith constants.For compilation purposes, writing the name is the same as writing the value.

If the named constant is used as the length specifier in a CHARACTER declaration, it must be enclosed in parentheses.

The name of a constant cannot appear as part of another constant, although it can appear as either the real or imaginary part of a complex constant.

You can only use the named constant within the scoping unit containing the defining PARAMETER statement.

Any named constant that appears in the initialization expression must have been defined previously in the same type declaration statement (or in a previous type declaration statement or PARAMETER statement), or made accessible by use or host association.

The use of parentheses is optional and can be controlled by using compiler option [no]altparam.

Example

The following example shows a type declaration statement specifying the PARAMETER attribute:

```
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
```

The following is an example of the PARAMETER statement:

```
REAL(4) PI, PIOV2

REAL(8) DPI, DPIOV2

LOGICAL FLAG

CHARACTER*(*) LONGNAME

PARAMETER (PI=3.1415927, DPI=3.141592653589793238D0)

PARAMETER (PIOV2=PI/2, DPIOV2=DPI/2)

PARAMETER (FLAG=.TRUE., LONGNAME='A STRING OF 25 CHARACTERS')
```

The following shows another example:

```
! implicit integer type
PARAMETER (nblocks = 10)
! implicit real type
IMPLICIT REAL (L-M)
PARAMETER (loads = 10.0, mass = 32.2)
! typed by PARAMETER statement
! Requires compiler option
PARAMETER mass = 47.3, pi = 3.14159
PARAMETER bigone = 'This constant is larger than forty characters'
! PARAMETER in attribute syntax
REAL, PARAMETER :: mass=47.3, pi=3.14159, loads=10.0, mass=32.2
```

See Also

- O to P
- DATA
- Type Declarations
- Compatible attributes
- Initialization Expressions
- IMPLICIT
- Alternative syntax for the PARAMETER statement
- altparam compiler option

PASSDIRKEYSQQ (W*32, W*64)

QuickWin Function: *Determines the behavior of direction and page keys in a QuickWin application.*

Module

USE IFQWIN

Syntax

`result = PASSDIRKEYSQQ (val)`

val (Input) INTEGER(4) or LOGICAL(4).

A value of .TRUE. causes direction and page keys to be input as normal characters (the PassDirKeys flag is turned on). A value of .FALSE. causes direction and page keys to be used for scrolling. The following constants, defined in IFQWIN.F90, can be used as integer arguments:

- PASS_DIR_FALSE - Turns off any special handling of direction keys. They are not passed to the program by GETCHARQQ.
- PASS_DIR_TRUE - Turns on special handling of direction keys. That is, they are passed to the program by GETCHARQQ.
- PASS_DIR_INSDEL - INSERT and DELETE are also passed to the program by GETCHARQQ
- PASS_DIR_CNTRLC - Only needed for a QuickWin application, but harmless if used with a Standard Graphics application that already passes CTRL+C.

  This value allows CTRL+C to be passed to a QuickWin program by GETCHARQQ if the following is true: the program must have removed the File menu EXIT item by using DELETEMENUQQ.

  This value also passes direction keys and INSERT and DELETE.

Results

The return value indicates the previous setting of the PassDirKeys flag.

The return data type is the same as the data type of *val*; that is, either INTEGER(4) or LOGICAL(4).

When the PassDirKeys flag is turned on, the mouse must be used for scrolling since the direction and page keys are treated as normal input characters.

The PASSDIRKEYSQQ function is meant to be used primarily with the GETCHARQQ and INCHARQQ functions. Do not use normal input statements (such as READ) with the PassDirKeys flag turned on, unless your program is prepared to interpret direction and page keys.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
use IFQWIN
logical*4 res
character*1 ch, ch1
Print *,"Type X to exit, S to scroll, D to pass Direction keys"
123 continue
ch = getcharqq( )
! check for escapes
! 0x00 0x?? is a function key
! 0xE0 0x?? is a direction key
if (ichar(ch) .eq. 0) then
  ch1 = getcharqq()
  print *,"function key follows escape = ",ichar(ch), "  ",ichar(ch1),"  ",ch1
  goto 123
else if (ichar(ch) .eq. 224) then
  ch1 = getcharqq()
  print *,"direction key follows escape = ",ichar(ch),"  ",ichar(ch1),"  ",ch1
  goto 123
else
  print *,ichar(ch),"  ",ch
  if(ch .eq. 'S') then
   res = passdirkeysqq(.false.)
   print *, "Entering Scroll mode ",res
  endif
  if(ch .eq. 'D')  then
   res = passdirkeysqq(.true.)
   print *, "Entering Direction keys mode ",res
  endif
```

```
  if(ch .ne. 'X')  go to 123
endif
end
```

The following example uses an integer constant as an argument to PASSDIRKEYSQQ:

```
c======================================================================
c
c dirkeys4.for
c
c======================================================================
c
c       Compile/Load Input Line for Standard Graphics Full Screen Window
c
c       ifort /libs:qwins dirkeys4.for
c
c       Compile/Load Input Line for QuickWin Graphics
c
c       ifort /libs:qwin dirkeys4.for
c
c  Program to illustrate how to get almost every character
c  from the keyboard in QuickWin or Standard Graphics mode.
c  Comment out the deletemenu line for Standard Graphics mode.
c
c  If you are doing a standard graphics application,
c  control C will come in as a Z'03' without further
c  effort.
c
c  In a QuickWin application,  The File menu Exit item must
c  be deleted, and PassDirKeysQQ called with PASS_DIR_CNTRLC
c  to get control C.
c
c======================================================================
      use IFQWIN
```

```
      integer(4) status
      character*1 key1,key2,ch1
      write(*,*) 'Initializing'
c-----don't do this for a Standard Grapics application
c     remove File menu Exit item.
      status = deletemenuqq(1,3) ! stop QuickWin from getting control C
c-----set up to pass all keys to window including control c.
      status  = passdirkeysqq(PASS_DIR_CNTRLC)
c=================================================================
c
c     read and print characters
c
c=================================================================
   10 key1 = getcharqq()
c-----first check for control+c
      if(ichar(key1) .eq. 3) then
        write(*,*) 'Control C Received'
        write(*,*) "Really want to quit?"
        write(*,*) "Type Y <cr> to exit, or any other char <cr> to continue."
        read(*,*) ch1
        if(ch1.eq."y" .or. ch1.eq."Y") goto 30
        goto 10
      endif
      if(ichar(key1).eq.0) then ! function key?
      key2 = getcharqq()
      write(*,15) ichar(key1),ichar(key2),key2
   15 format(1x,2i12,1x,a1,' function key')
      else
```

```
      if(ichar(key1).eq.224) then ! direction key?
      key2 = getcharqq()
      write(*,16) ichar(key1),ichar(key2),key2
   16 format(1x,2i12,1x,a1,' direction key')
      else
      write(*,20) key1,ichar(key1)  ! normal key
   20 format(1x,a1,i11)
      endif
      endif
      go to 10
   30 stop
      end
```

See Also

- O to P
- GETCHARQQ
- INCHARQQ

PAUSE

Statement: *Temporarily suspends program execution and lets you execute operating system commands during the suspension. The PAUSE statement is a deleted feature in Fortran 95; it was obsolescent in Fortran 90. Intel Fortran fully supports features deleted in Fortran 95.*

Syntax

`PAUSE [pause-code]`

pause-code (Optional) Is an optional message. It can be either of the following:

- A scalar character constant of type default character.
- A string of up to six digits; leading zeros are ignored. (Fortran 90 and FORTRAN 77 limit digits to five.)

If you specify *pause-code*, the PAUSE statement displays the specified message and then displays the default prompt.

If you do not specify *pause-code*, the system displays the following default message:

```
FORTRAN PAUSE
```

The following prompt is then displayed:

- On Windows* systems:

  ```
  Fortran Pause - Enter command<CR> or <CR> to continue.
  ```

- On Linux* and Mac OS* X systems:

  ```
  PAUSE prompt>
  ```

Effect on Windows* Systems

The program waits for input on `stdin`. If you enter a blank line, execution resumes at the next executable statement.

Anything else is treated as a DOS command and is executed by a system( ) call. The program loops, letting you execute multiple DOS commands, until a blank line is entered. Execution then resumes at the next executable statement.

Effect on Linux* and Mac OS* Systems

The effect of PAUSE differs depending on whether the program is a foreground or background process, as follows:

- If a program is a foreground process, the program is suspended until you enter the CONTINUE command. Execution then resumes at the next executable statement.

  Any other command terminates execution.

- If a program is a background process, the behavior depends on `stdin`, as follows:

  - If `stdin`is redirected from a file, the system displays the following (after the pause code and prompt):

    ```
    To continue from background, execute 'kill -15 n'
    ```

    In this message, `n`is the process id of the program.

  - If `stdin`is not redirected from a file, the program becomes a suspended background job, and you must specify `fg`to bring the job into the foreground. You can then enter a command to resume or terminate processing.

Example

The following examples show valid PAUSE statements:

```
PAUSE 701

PAUSE 'ERRONEOUS RESULT DETECTED'
```

The following shows another example:

```
CHARACTER*24 filename

PAUSE 'Enter DIR to see available files or press RETURN'   &

&' if you already know filename.'

READ(*,'(A\)') filename

OPEN(1, FILE=filename)

. . .
```

See Also

- O to P
- STOP
- SYSTEM
- Obsolescent and Deleted Language Features

PEEKCHARQQ

Run-Time Function: *Checks the keystroke buffer for a recent console keystroke and returns .TRUE. if there is a character in the buffer or .FALSE. if there is not.*

Module

USE IFCORE

Syntax

```
result = PEEKCHARQQ( )
```

Results

The result type is LOGICAL(4). The result is .TRUE. if there is a character waiting in the keyboard buffer; otherwise, .FALSE..

To find out the value of the key in the buffer, call GETCHARQQ. If there is no character waiting in the buffer when you call GETCHARQQ, GETCHARQQ waits until there is a character in the buffer. If you call PEEKCHARQQ first, you prevent GETCHARQQ from halting your process while it waits for a keystroke. If there is a keystroke, GETCHARQQ returns it and resets PEEKCHARQQ to .FALSE..

Compatibility

CONSOLE DLL LIB

Example

```
USE IFCORE
LOGICAL(4) pressed / .FALSE. /
DO WHILE (.NOT. pressed)
  WRITE(*,*) ' Press any key'
  pressed = PEEKCHARQQ ( )
END DO
END
```

See Also

- O to P
- GETCHARQQ
- GETSTRQQ
- FGETC
- GETC

PERROR

Run-Time Subroutine: *Sends a message to the standard error stream, preceded by a specified string, for the last detected error.*

Module

USE IFCORE

Syntax

CALL PERROR (*string*)

string	(Input) Character*(*). Message to precede the standard error message.

The string sent is the same as that given by GERROR.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFCORE
character*24 errtext
errtext = 'In my opinion, '
. . .
! any error message generated by errtext is
! preceded by 'In my opinion, '
Call PERROR (errtext)
```

See Also

- O to P
- GERROR
- IERRNO

PIE, PIE_W (W*32, W*64)

Graphics Functions: *Draw a pie-shaped wedge in the current graphics color.*

Module

USE IFQWIN

Syntax

```
result = PIE (i,x1,y1,x2,y2,x3,y3,x4,y4)
result = PIE_W (i, wx1, wy1, wx2, wy2, wx3, y3,wx4, wy4)
```

i	(Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in IFQWIN.F90): • $GFILLINTERIOR - Fills the figure using the current color and fill mask. • $GBORDER - Does not fill the figure.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of bounding rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of bounding rectangle.
x3, y3	(Input) INTEGER(2). Viewport coordinates of start vector.
x4, y4	(Input) INTEGER(2). Viewport coordinates of end vector.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of bounding rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of bounding rectangle.
wx3, wy3	(Input) REAL(8). Window coordinates of start vector.
wx4, wy4	(Input) REAL(8). Window coordinates of end vector.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0. If the pie is clipped or partially out of bounds, the pie is considered successfully drawn and the return is 1. If the pie is drawn completely out of bounds, the return is 0.

The border of the pie wedge is drawn in the current color set by SETCOLORRGB.

The PIE function uses the viewport-coordinate system. The center of the arc is the center of the bounding rectangle, which is specified by the viewport-coordinate points (*x1*, *y1*) and (*x2*, *y2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*).

The PIE_W function uses the window-coordinate system. The center of the arc is the center of the bounding rectangle specified by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*). The arc starts where it intersects an imaginary line extending from the center of the arc through (*wx3*, *wy3*). It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*wx4*, *wy4*).

The fill flag option $GFILLINTERIOR is equivalent to a subsequent call to FLOODFILLRGB using the center of the pie as the starting point and the current graphics color (set by SETCOLORRGB) as the fill color. If you want a fill color different from the boundary color, you cannot use the $GFILLINTERIOR option. Instead, after you have drawn the pie wedge, change the current color with SETCOLORRGB and then call FLOODFILLRGB. You must supply FLOODFILLRGB with an interior point in the figure you want to fill. You can get this point for the last drawn pie or arc by calling GETARCINFO.

If you fill the pie with FLOODFILLRGB, the pie must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE. The PIE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Pie routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Pie. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

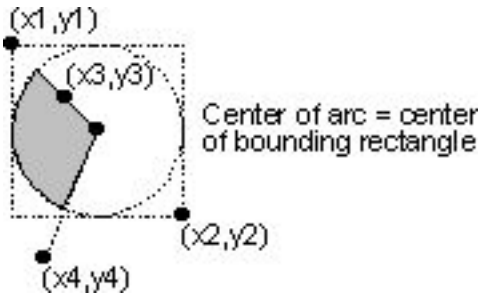
Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! build as Graphics App.
USE IFQWIN
INTEGER(2) status, dummy
INTEGER(2) x1, y1, x2, y2, x3, y3, x4, y4
x1 = 80; y1 = 50
x2 = 180; y2 = 150
x3 = 110; y3 = 80
x4 = 90; y4 = 180
status = SETCOLOR(INT2(4))
dummy = PIE( $GFILLINTERIOR, x1, y1, x2, y2, &
                             x3, y3, x4, y4)
END
```

This figure shows the coordinates used to define PIE and PIE_W:



See Also

- O to P
- SETCOLORRGB
- SETFILLMASK
- SETLINESTYLE
- FLOODFILLRGB
- GETARCINFO
- ARC
- ELLIPSE
- GRSTATUS
- LINETO
- POLYGON
- RECTANGLE

POINTER - Fortran 95/90

Statement and Attribute: *Specifies that an object is a pointer (a dynamic variable). A pointer does not contain data, but* `points` *to a scalar or array variable where data is stored. A pointer has no initial storage set aside for it; memory storage is created for the pointer as a program runs.*

Syntax

The POINTER attribute can be specified in a type declaration statement or a POINTER statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] POINTER [, *att-ls*] :: *ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]]...

Statement:

POINTER [::]*ptr*[(*d-spec*)][, *ptr*[(*d-spec*)]] ...

type-spec	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
ptr	Is the name of the pointer. The pointer cannot be declared with the INTENT or PARAMETER attributes.
d-spec	(Optional) Is a deferred-shape specification (: [, :] ...). Each colon represents a dimension of the array.

Description

No storage space is created for a pointer until it is allocated with an ALLOCATE statement or until it is assigned to a allocated target. A pointer must not be referenced or defined until memory is associated with it.

Each pointer has an association status, which tells whether the pointer is currently associated with a target object. When a pointer is initially declared, its status is undefined. You can use the ASSOCIATED intrinsic function to find the association status of a pointer if the pointer's association status is defined.

If the pointer is an array, and it is given the DIMENSION attribute elsewhere in the program, it must be declared as a deferred-shape array.

A pointer cannot be specified in an EQUIVALENCE or NAMELIST statement. A pointer in a DATA statement can only be associated with NULL().

Fortran 95/90 pointers are not the same as integer pointers. For more information, see the POINTER - Integer statement.

Example

The following example shows type declaration statements specifying the POINTER attribute:

```
TYPE(SYSTEM), POINTER :: CURRENT, LAST

REAL, DIMENSION(:,:), POINTER :: I, J, REVERSE
```

The following is an example of the POINTER statement:

```
TYPE(SYSTEM) :: TODAYS

POINTER :: TODAYS, A(:,:)
```

See also the examples POINTER.F90 and POINTER2.F90in the TUTORIAL sample programs.

The following shows another example:

```
REAL, POINTER :: arrow (:)
REAL, ALLOCATABLE, TARGET :: bullseye (:,:)
! The following statement associates the pointer with an unused
! block of memory.
ALLOCATE (arrow (1:8), STAT = ierr)
IF (ierr.eq.0) WRITE (*,'(/1x,a)') 'ARROW allocated'
arrow = 5.
WRITE (*,'(1x,8f8.0/)') arrow
ALLOCATE (bullseye (1:8,3), STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'BULLSEYE allocated'
bullseye = 1.
bullseye (1:8:2,2) = 10.
WRITE (*,'(1x,8f8.0)') bullseye
! The following association breaks the association with the first
! target, which being unnamed and unassociated with other pointers,
! becomes lost. ARROW acquires a new shape.
arrow => bullseye (2:7,2)
WRITE (*,'(/1x,a)') 'ARROW is repointed & resized, all the 5s are lost'
WRITE (*,'(1x,8f8.0)') arrow
NULLIFY (arrow)
IF (.NOT.ASSOCIATED(arrow)) WRITE (*,'(/a/)') ' ARROW is not pointed'
DEALLOCATE (bullseye, STAT = ierr)
IF (ierr.eq.0) WRITE (*,*) 'Deallocation successful.'
END
```

See Also

- O to P

- ALLOCATE
- ASSOCIATED
- DEALLOCATE
- NULLIFY
- TARGET
- Deferred-Shape Arrays
- Pointer Assignments
- Pointer Association
- Pointer Arguments
- NULL
- Integer POINTER statement
- Type Declarations
- Compatible attributes

POINTER - Integer

Statement: *Establishes pairs of objects and pointers, in which each pointer contains the address of its paired object. This statement is different from the Fortran 95/90 POINTERstatement.*

Syntax

```
POINTER (pointer,pointee) [,(pointer,pointee)] . . .
```

pointer	Is a variable whose value is used as the address of the *pointee*.
pointee	Is a variable; it can be an array name or array specification. It can also be a procedure named in an EXTERNAL statement or in a specific (non-generic) procedure interface block.

The following are *pointer* rules and behavior:

- Two pointers can have the same value, so pointer aliasing is allowed.
- When used directly, a pointer is treated like an integer variable. On IA-32 architecture, a pointer occupies one numeric storage unit, so it is a 32-bit quantity (INTEGER(4)). On Intel® 64 architecture and IA-64 architecture, a pointer occupies two numeric storage units, so it is a 64-bit quantity (INTEGER(8)).
- A pointer cannot be a pointee.
- A pointer cannot appear in an ASSIGN statement and cannot have the following attributes:

ALLOCATABLE	INTRINSIC	POINTER
EXTERNAL	PARAMETER	TARGET

A pointer can appear in a DATA statement with integer literals only.

- Integers can be converted to pointers, so you can point to absolute memory locations.
- A pointer variable cannot be declared to have any other data type.
- A pointer cannot be a function return value.
- You can give values to pointers by doing the following:
  - Retrieve addresses by using the LOC intrinsic function (or the %LOC built-in function)
  - Allocate storage for an object by using the MALLOC intrinsic function (or by using malloc(3f) on Linux* and Mac OS* X systems)

  For example:

```
Using %LOC:              Using MALLOC:
INTEGER I(10)                 INTEGER I(10)
INTEGER I1(10) /10*10/        POINTER (P,I)
POINTER (P,I)                 P = MALLOC(40)
P = %LOC(I1)                  I = 10
I(2) = I(2) + 1               I(2) = I(2) + 1
```

- The value in a pointer is used as the pointee's base address.

The following are *pointee* rules and behavior:

- A pointee is not allocated any storage. References to a pointee look to the current contents of its associated pointer to find the pointee's base address.
- A pointee cannot be data-initialized or have a record structure that contains data-initialized fields.
- A pointee can appear in only one integer POINTER statement.
- A pointee array can have fixed, adjustable, or assumed dimensions.
- A pointee cannot appear in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement, and it cannot have the following attributes:

ALLOCATABLE	OPTIONAL	SAVE

AUTOMATIC	PARAMETER	STATIC
INTENT	POINTER	

- A pointee cannot be:
  - A dummy argument
  - A function return value
  - A record field or an array element
  - Zero-sized
  - An automatic object
  - The name of a generic interface block
- If a pointee is of derived type, it must be of sequence type.

Example

```
POINTER (p, k)
INTEGER j(2)
! This has the same effect as j(1) = 0, j(2) = 5
p = LOC(j)
k = 0
p = p + SIZEOF(k) ! 4 for 4-byte integer
k = 5
```

See Also

- O to P
- LOC
- MALLOC
- FREE

POLYBEZIER, POLYBEZIER_W (W*32, W*64)

Graphics Functions: *Draw one or more Bezier curves.*

Module

USE IFQWIN

Syntax

`result = POLYBEZIER (`*`ppoints`*`,`*`cpoints`*`)`

`result = POLYBEZIER_W (`*`wppoints`*`,`*`cpoints`*`)`

`ppoints` (Input) Derived type `xycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

`cpoints` (Input) INTEGER(2). Number of points in *`ppoints`* or *`wppoints`*.

`wppoints` (Input) Derived type `wxycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `wxycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx
  REAL(8) wy
END TYPE wxycoord
```

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

A Bezier curve is based on fitting a cubic curve to four points. The first point is the starting point, the next two points are control points, and last point is the ending point. The starting point must be given for the first curve; subsequent curves use the ending point of the previous curve as their starting point. So, *cpoints* should contain 4 for one curve, 7 for 2 curves, 10 for 3 curves, and so forth.

POLYBEZIER does not use or change the current graphics position.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
Program Bezier
use IFQWIN
! Shows how to use POLYBEZIER, POLYBEZIER_W,
! POLYBEZIERTO, and POLYBEZIERTO_W,
TYPE(xycoord)  lppoints(31)
TYPE(wxycoord) wlppoints(31)
TYPE(xycoord)  xy
TYPE(wxycoord) wxy
integer(4) i
integer(2) istat, orgx, orgy
real(8) worgx, worgy
i = setcolorrgb(Z'00FFFFFF') ! graphic to black
i = settextcolorrgb(Z'00FFFFFF') ! text to black
i = setbkcolorrgb(Z'00000000') ! background to white
call clearscreen($GCLEARSCREEN)
orgx = 20
orgy = 20
lppoints(1).xcoord = 1+orgx
lppoints(1).ycoord = 1+orgy
lppoints(2).xcoord = 30+orgx
lppoints(2).ycoord = 120+orgy
lppoints(3).xcoord = 150+orgx
lppoints(3).ycoord = 60+orgy
lppoints(4).xcoord = 180+orgx
lppoints(4).ycoord = 180+orgy
istat = PolyBezier(lppoints, 4)
! Show tangent lines
```

```
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto(lppoints(i).xcoord,lppoints(i).ycoord,xy)
istat = lineto(lppoints(i+1).xcoord,lppoints(i+1).ycoord)
end do
read(*,*)
worgx = 50.0
worgy = 50.0
wlppoints(1).wx = 1.0+worgx
wlppoints(1).wy = 1.0+worgy
wlppoints(2).wx = 30.0+worgx
wlppoints(2).wy = 120.0+worgy
wlppoints(3).wx = 150.0+worgx
wlppoints(3).wy = 60.0+worgy
wlppoints(4).wx = 180.0+worgx
wlppoints(4).wy = 180.0+worgy
i = setcolorrgb(Z'000000FF') ! graphic to red
istat = PolyBezier_W(wlppoints, 4)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto_w(wlppoints(i).wx,wlppoints(i).wy,wxy)
```

```
istat = lineto_w(wlppoints(i+1).wx,wlppoints(i+1).wy)
end do
read(*,*)
orgx = 80
orgy = 80
! POLYBEZIERTO uses the current graphics position
! as its initial starting point so we start the
! array with the first first control point.
! lppoints(1).xcoord = 1+orgx ! need to move to this
! lppoints(1).ycoord = 1+orgy
lppoints(1).xcoord = 30+orgx
lppoints(1).ycoord = 120+orgy
lppoints(2).xcoord = 150+orgx
lppoints(2).ycoord = 60+orgy
lppoints(3).xcoord = 180+orgx
lppoints(3).ycoord = 180+orgy
i = setcolorrgb(Z'0000FF00') ! graphic to green
call moveto(1+orgx,1+orgy,xy)
istat = PolyBezierTo(lppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto(1+orgx,1+orgy,xy)
istat = lineto(lppoints(1).xcoord,lppoints(1).ycoord)
call moveto(lppoints(2).xcoord,lppoints(2).ycoord,xy)
istat = lineto(lppoints(3).xcoord,lppoints(3).ycoord)
```

```
read(*,*)
worgx = 110.0
worgy = 110.0
!wlppoints(1).wx = 1.0+worgx
!wlppoints(1).wy = 1.0+worgy
wlppoints(1).wx = 30.0+worgx
wlppoints(1).wy = 120.0+worgy
wlppoints(2).wx = 150.0+worgx
wlppoints(2).wy = 60.0+worgy
wlppoints(3).wx = 180.0+worgx
wlppoints(3).wy = 180.0+worgy
call moveto_w(1.0+worgx,1.0+worgy,wxy)
i = setcolorrgb(Z'00FF0000') ! graphic to blue
istat = PolyBezierTo_W(wlppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto_w(1.0+worgx,1.0+worgy,wxy)
istat = lineto_w(wlppoints(1).wx,wlppoints(1).wy)
call moveto_w(wlppoints(2).wx,wlppoints(2).wy,wxy)
istat = lineto_w(wlppoints(3).wx,wlppoints(3).wy)
read(*,*)
END PROGRAM Bezier
```

See Also

- O to P
- POLYBEZIERTO, POLYBEZIERTO_W

POLYBEZIERTO, POLYBEZIERTO_W (W*32, W*64)

Graphics Functions: *Draw one or more Bezier curves.*

Module

USE IFQWIN

Syntax

`result = POLYBEZIERTO (`*`ppoints`*`,`*`cpoints`*`)`

`result = POLYBEZIERTO_W (`*`wppoints`*`,`*`cpoints`*`)`

ppoints (Input) Derived type `xycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints (Input) INTEGER(2). Number of points in *ppoints* or *wppoints*.

wppoints (Input) Derived type `wxycoord`. Array of derived types defining the endpoints and the control points for each Bezier curve. The derived type `wxycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE wxycoord
  REAL(8) wx
  REAL(8) wy
END TYPE wxycoord
```

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

A Bezier curve is based on fitting a cubic curve to four points. The first point is the starting point, the next two points are control points, and last point is the ending point. The starting point is the current graphics position as set by MOVETO for the first curve; subsequent curves use the ending point of the previous curve as their starting point. So, *cpoints* should contain 3 for one curve, 6 for 2 curves, 9 for 3 curves, and so forth.

POLYBEZIERTO moves the current graphics position to the ending point of the last curve drawn.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
Program Bezier
use IFQWIN
! Shows how to use POLYBEZIER, POLYBEZIER_W,
! POLYBEZIERTO, and POLYBEZIERTO_W,
TYPE(xycoord)  lppoints(31)
TYPE(wxycoord) wlppoints(31)
TYPE(xycoord)  xy
TYPE(wxycoord) wxy
integer(4) i
integer(2) istat, orgx, orgy
real(8) worgx, worgy
i = setcolorrgb(Z'00FFFFFF') ! graphic to black
i = settextcolorrgb(Z'00FFFFFF') ! text to black
i = setbkcolorrgb(Z'00000000') ! background to white
call clearscreen($GCLEARSCREEN)
orgx = 20
orgy = 20
lppoints(1).xcoord = 1+orgx
lppoints(1).ycoord = 1+orgy
lppoints(2).xcoord = 30+orgx
lppoints(2).ycoord = 120+orgy
lppoints(3).xcoord = 150+orgx
lppoints(3).ycoord = 60+orgy
lppoints(4).xcoord = 180+orgx
lppoints(4).ycoord = 180+orgy
istat = PolyBezier(lppoints, 4)
! Show tangent lines
```

```
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto(lppoints(i).xcoord,lppoints(i).ycoord,xy)
istat = lineto(lppoints(i+1).xcoord,lppoints(i+1).ycoord)
end do
read(*,*)
worgx = 50.0
worgy = 50.0
wlppoints(1).wx = 1.0+worgx
wlppoints(1).wy = 1.0+worgy
wlppoints(2).wx = 30.0+worgx
wlppoints(2).wy = 120.0+worgy
wlppoints(3).wx = 150.0+worgx
wlppoints(3).wy = 60.0+worgy
wlppoints(4).wx = 180.0+worgx
wlppoints(4).wy = 180.0+worgy
i = setcolorrgb(Z'000000FF') ! graphic to red
istat = PolyBezier_W(wlppoints, 4)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
do i = 1,4,2
call moveto_w(wlppoints(i).wx,wlppoints(i).wy,wxy)
```

```
istat = lineto_w(wlppoints(i+1).wx,wlppoints(i+1).wy)
end do
read(*,*)
orgx = 80
orgy = 80
! POLYBEZIERTO uses the current graphics position
! as its initial starting point so we start the
! array with the first first control point.
! lppoints(1).xcoord = 1+orgx ! need to move to this
! lppoints(1).ycoord = 1+orgy
lppoints(1).xcoord = 30+orgx
lppoints(1).ycoord = 120+orgy
lppoints(2).xcoord = 150+orgx
lppoints(2).ycoord = 60+orgy
lppoints(3).xcoord = 180+orgx
lppoints(3).ycoord = 180+orgy
i = setcolorrgb(Z'0000FF00') ! graphic to green
call moveto(1+orgx,1+orgy,xy)
istat = PolyBezierTo(lppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto(1+orgx,1+orgy,xy)
istat = lineto(lppoints(1).xcoord,lppoints(1).ycoord)
call moveto(lppoints(2).xcoord,lppoints(2).ycoord,xy)
istat = lineto(lppoints(3).xcoord,lppoints(3).ycoord)
```

```
read(*,*)
worgx = 110.0
worgy = 110.0
! wlppoints(1).wx = 1.0+worgx
! wlppoints(1).wy = 1.0+worgy
wlppoints(1).wx = 30.0+worgx
wlppoints(1).wy = 120.0+worgy
wlppoints(2).wx = 150.0+worgx
wlppoints(2).wy = 60.0+worgy
wlppoints(3).wx = 180.0+worgx
wlppoints(3).wy = 180.0+worgy
call moveto_w(1.0+worgx,1.0+worgy,wxy)
i = setcolorrgb(Z'00FF0000') ! graphic to blue
istat = PolyBezierTo_W(wlppoints, 3)
! Show tangent lines
! A bezier curve is tangent to the line
! from the begin point to the first control
! point.   It is also tangent to the line from
! the second control point to the end point.
call moveto_w(1.0+worgx,1.0+worgy,wxy)
istat = lineto_w(wlppoints(1).wx,wlppoints(1).wy)
call moveto_w(wlppoints(2).wx,wlppoints(2).wy,wxy)
istat = lineto_w(wlppoints(3).wx,wlppoints(3).wy)
read(*,*)
END PROGRAM Bezier
```

See Also

- O to P
- POLYBEZIER, POLYBEZIER_W
- MOVETO, MOVETO_W

POLYGON, POLYGON_W (W*32, W*64)

Graphics Functions: *Draw a polygon using the current graphics color, logical write mode, and line style.*

Module

USE IFQWIN

Syntax

`result = POLYGON (control,ppoints,cpoints)`

`result = POLYGON_W (control,wppoints,cpoints)`

control (Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in `IFQWIN.F90`):

- $GFILLINTERIOR - Draws a solid polygon using the current color and fill mask.
- $GBORDER - Draws the border of a polygon using the current color and line style.

ppoints (Input) Derived type `xycoord`. Array of derived types defining the polygon vertices in viewport coordinates. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:

```
TYPE xycoord
  INTEGER(2) xcoord
  INTEGER(2) ycoord
END TYPE xycoord
```

cpoints (Input) INTEGER(2). Number of polygon vertices.

wppoints (Input) Derived type wxycoord. Array of derived types defining the polygon vertices in window coordinates. The derived type wxycoordis defined in IFQWIN.F90 as follows:

```
TYPE wxycoord
  REAL(8) wx
  REAL(8) wy
END TYPE wxycoord
```

Results

The result type is INTEGER(2). The result is nonzero if anything is drawn; otherwise, 0.

The border of the polygon is drawn in the current graphics color, logical write mode, and line style, set with SETCOLORRGB, SETWRITEMODE, and SETLINESTYLE, respectively. The POLYGON routine uses the viewport-coordinate system (expressed in xycoord derived types), and the POLYGON_W routine uses real-valued window coordinates (expressed in wxycoordtypes).

The arguments *ppoints* and *wppoints* are arrays whose elements are xycoordor wxycoord derived types. Each element specifies one of the polygon's vertices. The argument *cpoints* is the number of elements (the number of vertices) in the *ppoints* or *wppoints* array.

Note that POLYGON draws between the vertices in their order in the array. Therefore, when drawing outlines, skeletal figures, or any other figure that is not filled, you need to be careful about the order of the vertices. If you don't want lines between some vertices, you may need to repeat vertices to make the drawing backtrack and go to another vertex to avoid drawing across your figure. Also, POLYGON draws a line from the last specified vertex back to the first vertex.

If you fill the polygon using FLOODFILLRGB, the polygon must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE. The POLYGON routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Polygon routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Polygon. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics App.
 !
 ! Draw a skeletal box
     USE IFQWIN
     INTEGER(2) status
     TYPE (xycoord) poly(12)
 ! Set up box vertices in order they will be drawn, &
 ! repeating some to avoid unwanted lines across box
     poly(1)%xcoord = 50
     poly(1)%ycoord = 80
     poly(2)%xcoord = 85
     poly(2)%ycoord = 35
     poly(3)%xcoord = 185
     poly(3)%ycoord = 35
     poly(4)%xcoord = 150
     poly(4)%ycoord = 80
     poly(5)%xcoord = 50
     poly(5)%ycoord = 80
     poly(6)%xcoord = 50
     poly(6)%ycoord = 180
     poly(7)%xcoord = 150
     poly(7)%ycoord = 180
     poly(8)%xcoord = 185
     poly(8)%ycoord = 135
     poly(9)%xcoord = 185
     poly(9)%ycoord = 35
     poly(10)%xcoord = 150
```

```
poly(10)%ycoord = 80
poly(11)%xcoord = 150
poly(11)%ycoord = 180
poly(12)%xcoord = 150
poly(12)%ycoord = 80
status = SETCOLORRGB(Z'0000FF')
status = POLYGON($GBORDER, poly, INT2(12))
END
```

See Also

- O to P
- SETCOLORRGB
- SETFILLMASK
- SETLINESTYLE
- FLOODFILLRGB
- GRSTATUS
- LINETO
- RECTANGLE
- SETWRITEMODE

POLYLINEQQ (W*32, W*64)

Graphics Function: *Draws a line between each successive x, y point in a given array.*

Module

USE IFQWIN

Syntax

`result = POLYLINEQQ (points,cnt)`

points	(Input) An array of DF_POINTobjects. The derived type DF_POINT is defined in IFQWIN.F90 as:
	`type DF_POINT`
	`  sequence`
	`  integer(4) x`
	`  integer(4) y`
	`end type DF_POINT`

cnt	(Input) INTEGER(4). Number of elements in the *points* array.

Results

The result type is INTEGER(4). The result is a nonzero value if successful; otherwise, zero.

POLYLINEQQ uses the viewport-coordinate system.

The lines are drawn using the current graphics color, logical write mode, and line style. The graphics color is set with SETCOLORRGB, the write mode with SETWRITEMODE, and the line style with SETLINESTYLE.

The current graphics position is not used or changed as it is in the LINETO function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS

Example

```
! Build for QuickWin or Standard Graphics
USE IFQWIN
TYPE(DF_POINT) points(12)
integer(4) result
integer(4) cnt, i
! load the points
do i = 1,12,2
  points(i).x =20*i
  points(i).y =10
  points(i+1).x =20*i
  points(i+1).y =60
end do
! A sawtooth pattern will appear in the upper left corner
result = POLYLINEQQ(points, 12)
end
```

See Also

- O to P
- LINETO
- LINETOAREX
- SETCOLORRGB
- SETLINESTYLE
- SETWRITEMODE

POPCNT

Elemental Intrinsic Function (Generic):
Returns the number of 1 bits in the integer argument.

Syntax

```
result = POPCNT (i)
```

i (Input) Must be of type integer or logical.

Results

The result type is the same as *i*. The result value is the number of 1 bits in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPCNT(I) is 5.

POPPAR

Elemental Intrinsic Function (Generic):
Returns the parity of the integer argument.

Syntax

```
result = POPPAR (i)
```

i (Input) Must be of type integer or logical.

Results

The result type is the same as *i*. The result value is 1 if there are an odd number of 1 bits in the binary representation of the integer I. The result value is zero if there are an even number.

POPPAR(*i*) is the same as 1 .AND. POPCNT(*i*).

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

If the value of I is B'0...00011010110', the value of POPPAR(I) is 1.

PRECISION

Inquiry Intrinsic Function (Generic): *Returns the decimal precision in the model representing real numbers with the same kind parameter as the argument.*

Syntax

```
result = PRECISION (x)
```

x	(Input) Must be of type real or complex; it can be scalar or array valued.

Results

The result is a scalar of type default integer. The result has the value INT((DIGITS(*x*) - 1) * LOG10(RADIX(*x*))). If RADIX(*x*) is an integral power of 10, 1 is added to the result.

Example

If X is a REAL(4) value, PRECISION(X) has the value 6. The value 6 is derived from INT ((24-1) * LOG10 (2.)) = INT (6.92...). For more information on the model for REAL(4), see Model for Real Data.

PREFETCH and NOPREFETCH

General Compiler Directives: *PREFETCH enables a data prefetch from memory. Prefetching data can minimize the effects of memory latency. NOPREFETCH (the default) disables data prefetching. These directives affect the heuristics used in the compiler.*

Syntax

```
cDEC$ PREFETCH [var1[: hint1[: distance1]] [,var2[: hint2[: distance2]]]...]
```

```
cDEC$ NOPREFETCH [var1[,var2]...]
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
var	Is an optional memory reference.

hint	Is an optional integer initialization expression with the value 0, 1, 2, or 3. These are the same as the values for *hint* in the intrinsic subroutine MM_PREFETCH. To use this argument, you must also specify *var*.
distance	Is an optional integer initialization expression with a value greater than 0. It indicates the number of loop iterations to perform before the prefetch. To use this argument, you must also specify *var* and *hint*.

To use these directives, compiler option O2 or O3 must be set.

This directive affects the DO loop it precedes.

If you specify PREFETCH with no arguments, all arrays accessed in the DO loop will be prefetched.

If a loop includes expression A(j), placing *c*DEC$ PREFETCH A in front of the loop instructs the compiler to insert prefetches for A(j + d) within the loop. The d is determined by the compiler.

Example

```
cDEC$ NOPREFETCH c
cDEC$ PREFETCH a
do i = 1, m
  b(i) = a(c(i)) + 1
enddo
```

The following example is valid on IA-64 architecture:

```
          sum = 0.d0
          do j=1,lastrow-firstrow+1
             i = rowstr(j)
             iresidue = mod( rowstr(j+1)-i, 8 )
             sum = 0.d0
CDEC$ NOPREFETCH a,p,colidx
             do k=i,i+iresidue-1
                sum = sum +  a(k)*p(colidx(k))
             enddo
CDEC$ NOPREFETCH colidx
CDEC$ PREFETCH a:1:40
CDEC$ PREFETCH p:1:20
             do k=i+iresidue, rowstr(j+1)-8, 8
                sum = sum + a(k  )*p(colidx(k  ))
     &                    + a(k+1)*p(colidx(k+1)) + a(k+2)*p(colidx(k+2))
     &                    + a(k+3)*p(colidx(k+3)) + a(k+4)*p(colidx(k+4))
     &                    + a(k+5)*p(colidx(k+5)) + a(k+6)*p(colidx(k+6))
     &                    + a(k+7)*p(colidx(k+7))
             enddo
          q(j) = sum
          enddo
```

See Also

- M to N
- O to P
- MM_PREFETCH
- O

Optimizing Applications: Prefetching Support

PRESENT

Inquiry Intrinsic Function (Generic): *Returns whether or not an optional dummy argument is present, that is, whether it has an associated actual argument.*

Syntax

```
result = PRESENT (a)
```

a	(Input) Must be an argument of the current procedure and must have the OPTIONAL attribute. An explicit interface for the current procedure must be visible to its caller; for more information, see Procedure Interfaces.

Results

The result is a scalar of type default logical. The result is .TRUE. if *a* is present; otherwise, the result is .FALSE..

Example

Consider the following:

```
MODULE MYMOD
CONTAINS
SUBROUTINE CHECK (X, Y)
  REAL X, Z
  REAL, OPTIONAL :: Y
  ...
  IF (PRESENT (Y)) THEN
    Z = Y
  ELSE
     Z = X * 2
  END IF
END SUBROUTINE CHECK
END MODULE MYMOD
...
USE MYMOD
CALL CHECK (15.0, 12.0) ! Causes Z to be set to 12.0
CALL CHECK (15.0) ! Causes Z to be set to 30.0
```

The following shows another example:

```
CALL who( 1, 2 ) ! prints "A present" "B present"
CALL who( 1 ) ! prints "A present"
CALL who( b = 2 ) ! prints "B present"
CALL who( ) ! prints nothing
CONTAINS
  SUBROUTINE who( a, b )
    INTEGER(4), OPTIONAL :: a, b
    IF (PRESENT(a)) PRINT *,'A present'
    IF (PRESENT(b)) PRINT *,'B present'
  END SUBROUTINE who
END
```

See Also

- O to P
- OPTIONAL
- Optional Arguments

PRINT

Statement: *Displays output on the screen. TYPE is a synonym for PRINT. All forms and rules for the PRINT statement also apply to the TYPE statement.*

Syntax

The PRINT statement is the same as a formatted, sequential WRITE statement, except that the PRINT statement must never transfer data to user-specified I/O units. You can override this restriction by using an environment variable. For more information, see *Building Applications*: Logical Devices.

A PRINT statement takes one of the following forms:

Formatted:

```
PRINT form[, io-list]
```

Formatted - List-Directed:

```
PRINT *[, io-list]
```

Formatted - Namelist:

```
PRINT nml
```

form	Is the nonkeyword form of a format specifier (no FMT=).
io-list	Is an I/O list.
*	Is the format specifier indicating list-directed formatting.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist formatting.

Example

In the following example, one record (containing four fields of data) is printed to the implicit output device:

```
      CHARACTER*16 NAME, JOB

      PRINT 400, NAME, JOB

400   FORMAT ('NAME=', A, 'JOB=', A)
```

The following shows another example:

```
!   The following statements are equivalent:

    PRINT     '(A11)', 'Abbottsford'

    WRITE (*, '(A11)') 'Abbottsford'

    TYPE      '(A11)', 'Abbottsford'
```

See Also

- O to P
- PUTC
- READ
- WRITE
- FORMAT
- Data Transfer I/O Statements
- File Operation I/O Statements
- PRINT as a value in CLOSE

PRIVATE Statement

Statement and Attribute: *Specifies that entities in a module can be accessed only within the module itself.*

Syntax

The PRIVATE attribute can be specified in a type declaration statement or a PRIVATE statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,]  PRIVATE [, att-ls] :: entity[, entity]...
```

Statement:

```
PRIVATE [[::] entity[, entity] ...]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is one of the following: • A variable name • A procedure name • A derived type name • A named constant • A namelist group name In statement form, an entity can also be a generic identifier (a generic name, defined operator, or defined assignment).

Description

The PRIVATE attribute can only appear in the scoping unit of a module.

Only one PRIVATE statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module.

If no PRIVATE statements are specified in a module, the default is PUBLIC accessibility. Entities with PUBLIC accessibility can be accessed from outside the module by means of a USE statement.

If a derived type is declared PRIVATE in a module, its components are also PRIVATE. The derived type and its components are accessible to any subprograms within the defining module through host association, but they are not accessible from outside the module.

If the derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

Example

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C
INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C
    DIMENSION LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following derived-type declaration statement indicates that the type is restricted to the module:

```
TYPE, PRIVATE  :: DATA
  ...
END TYPE DATA
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

The following shows another example:

```
 !LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 !The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
        REAL,INTENT(OUT) z
        CALL SQUARE(x,y)
        z = SQRT(x + y)
        RETURN
     END SUBROUTINE
     SUBROUTINE SQUARE(x1,y1)
        REAL x1,y1
        x1 = x1**2
        y1 = y1**2
        RETURN
     END SUBROUTINE
    END MODULE
```

See Also

- O to P
- MODULE
- PUBLIC
- TYPE
- Defining Generic Names for Procedures
- USE
- Use and Host Association

- Type Declarations
- Compatible attributes

PRIVATE Clause

Parallel Directive Clause: *Declares specified variables to be private to each thread in a team.*

Syntax

```
PRIVATE (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

The following occurs when variables are declared in a PRIVATE clause:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the construct and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

See Also

- O to P

Building Applications: Debugging Shared Variables

Optimizing Applications: OpenMP* Directives and Clauses Summary

Optimizing Applications: PRIVATE, FIRSTPRIVATE, and LASTPRIVATE Clauses

Optimizing Applications: REDUCTION Clause

Optimizing Applications: Data Scope Attribute Clauses Overview

Optimizing Applications: Parallel Region Directives

Optimizing Applications: Worksharing Construct Directives

PRODUCT

Transformational Intrinsic Function (Generic):
Returns the product of all the elements in an entire array or in a specified dimension of an array.

Syntax

```
result = PRODUCT (array[,dim] [,mask])
```

array	(Input) Must be an array of type integer, real, or complex.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If PRODUCT(*array*) is specified, the result is the product of all elements of *array*. If *array* has size zero, the result is 1.
- If PRODUCT(*array*, MASK= *mask*) is specified, the result is the product of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is 1.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as PRODUCT(*array*[,MASK= *mask*]).
- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- The value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of PRODUCT(*array*, *dim*[, *mask*]) is equal to PRODUCT(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK= *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)]).

Example

PRODUCT ((/2, 3, 4/)) returns the value 24 (the product of 2 * 3 * 4). PRODUCT ((/2, 3, 4/), DIM=1) returns the same result.

PRODUCT (C, MASK=C .LT. 0.0) returns the product of the negative elements of C.

A is the array

```
[ 1  4  7 ]
[ 2  3  5 ].
```

PRODUCT (A, DIM=1) returns the value (2, 12, 35), which is the product of all elements in each column. 2 is the product of 1 * 2 in column 1. 12 is the product of 4 * 3 in column 2, and so forth.

PRODUCT (A, DIM=2) returns the value (28, 30), which is the product of all elements in each row. 28 is the product of 1 * 4 * 7 in row 1. 30 is the product of 2 * 3 * 5 in row 2.

If *array* has shape (2, 2, 2), *mask* is omitted, and *dim* is 1, the result is an array result with shape (2, 2) whose elements have the following values.

Resultant array element	Value
result(1, 1)	*array*(1, 1, 1) * *array*(2, 1, 1)
result(2, 1)	*array*(1, 2, 1) * *array*(2, 2, 1)
result(1, 2)	*array*(1, 1, 2) * *array*(2, 1, 2)
result(2, 2)	*array*(1, 2, 2) * *array*(2, 2, 2)

The following shows another example:

```
INTEGER array (2, 3)
INTEGER AR1(3), AR2(2)
array = RESHAPE((/1, 4, 2, 5, 3, 6/),(/2,3/))
! array is  1 2 3
!           4 5 6
AR1 = PRODUCT(array, DIM = 1) ! returns [ 4 10 18 ]
AR2 = PRODUCT(array, MASK = array .LT. 6, DIM = 2)
                                  ! returns [ 6 20 ]
END
```

See Also

- O to P
- SUM

PROGRAM

Statement: *Identifies the program unit as a main program and gives it a name.*

Syntax

```
[PROGRAM name]
   [specification-part]
   [execution-part]
[CONTAINS
   internal-subprogram-part]
END[PROGRAM [name]]
```

name — Is the name of the program.

specification-part — Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- PUBLIC and PRIVATE (or their equivalent attributes)

	An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.
execution-part	Is one or more executable constructs or statements, except for ENTRY or RETURN statements.
internal-subprogram-part	Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

The PROGRAM statement is optional. Within a program unit, a PROGRAM statement can be preceded only by comment lines or an OPTIONS statement.

The END statement is the only required part of a program. If a name follows the END statement, it must be the same as the name specified in the PROGRAM statement.

The program name is considered global and must be unique. It cannot be the same as any local name in the main program or the name of any other program unit, external procedure, or common block in the executable program.

A main program must not reference itself (either directly or indirectly).

Example

The following is an example of a main program:

```
PROGRAM TEST
  INTEGER C, D, E(20,20)     ! Specification part
  CALL SUB_1                 ! Executable part
...
CONTAINS
  SUBROUTINE SUB_1           ! Internal subprogram
  ...
  END SUBROUTINE SUB_1
END PROGRAM TEST
```

The following shows another example:

```
PROGRAM MyProg
PRINT *, 'hello world'
END
```

See Also

- O to P

Building Applications: Building Applications from Microsoft* Visual Studio* .NET Overview

PROTECTED

Statement and Attribute: *Specifies limitations on the use of module entities.*

Syntax

The PROTECTED attribute can be specified in a type declaration statement or a PROTECTED statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls, ] PROTECTED [, att-ls] :: entity[, entity] ...
```

Statement:

```
PROTECTED [::]entity[, entity] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is the name of an entity in a module.

The PROTECTED attribute can only appear in the specification part of a module.

The PROTECTED attribute can only be specified for a named variable that is not in a common block.

A non-pointer object that has the PROTECTED attribute and is accessed by use association can not appear in a variable definition or as the target in a pointer assignment statement.

A pointer object that has the PROTECTED attribute and is accessed by use association must not appear as any of the following:

- A *pointer-object* in a NULLIFY statement
- A *pointer-object* in a pointer assignment statement

- An *object* in an ALLOCATE or DEALLOCATE statement
- An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

The following restrictions apply outside of the module in which the entity has been given the PROTECTED attribute:

- A non-pointer entity may not be defined or redefined.
- A pointer entity may not have its association status changed through the pointer.

Example

The following example shows a type declaration statement specifying the PROTECTED attribute:

```
INTEGER, PROTECTED :: D, E
```

Consider the following example:

```
MODULE counter_mod
  INTEGER, PROTECTED :: current = 0
  CONTAINS
  INTEGER FUNCTION next()
  current = current + 1    ! current can be modified here
  next = current
  RETURN
  END FUNCTION next
END MODULE counter_mod
PROGRAM test_counter
USE counter_mod
PRINT *, next( )             ! Prints 1
current = 42                 ! Error: variable is protected
END PROGRAM test_counter
```

See Also

- O to P
- Modules and Module Procedures

- Type Declarations
- Compatible attributes
- Pointer Assignments

PSECT

General Compiler Directive: *Modifies characteristics of a common block.*

Syntax

```
cDEC$ PSECT /common-name/ a[,a] ...
```

c — Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

common-name — Is the name of the common block. The slashes (/) are required.

a — Is one of the following:

- ALIGN= *val* or ALIGN= *keyword*

  Specifies minimum alignment for the common block. ALIGN only has an effect when specified on Windows* and Linux* systems.

  The *val* is a constant ranging from 0 through 6 on Windows* systems and 0 through 4 on Linux* systems. The specified number is interpreted as a power of 2. The value of the expression is the alignment in bytes.

  The *keyword* is one of the following:

Keyword	Equivalent to *val*
BYTE	0
WORD	1
LONG	2
QUAD	3
OCTA	4
PAGE[1]	i32: 12

Keyword	Equivalent to *val*
	i64: 13
[1]Range is 0 to 13 except on L*X32 , where the range is 0 to 12.	

- [NO]WRT

  Determines whether the contents of a common block can be modified during program execution.

If one program unit changes one or more characteristics of a common block, all other units that reference that common block must also change those characteristics in the same way.

The defaults are ALIGN=OCTA and WRT.

PUBLIC

Statement and Attribute: *Specifies that entities in a module can be accessed from outside the module (by specifying a USE statement).*

Syntax

The PUBLIC attribute can be specified in a type declaration statement or a PUBLIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,]  PUBLIC [, att-ls] :: entity [, entity]...
```

Statement:

```
PUBLIC [[::] entity [, entity] ...]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
entity	Is one of the following:

- A variable name
- A procedure name
- A derived type name
- A named constant

- A namelist group name

In statement form, an entity can also be a generic identifier (a generic name, defined operator, or defined assignment).

Description

The PUBLIC attribute can only appear in the scoping unit of a module.

Only one PUBLIC statement without an entity list is permitted in the scoping unit of a module; it sets the default accessibility of all entities in the module.

If no PRIVATE statements are specified in a module, the default is PUBLIC accessibility.

If a derived type is declared PUBLIC in a module, but its components are declared PRIVATE, any scoping unit accessing the module though use association (or host association) can access the derived-type definition, but not its components.

If a module procedure has a dummy argument or a function result of a type that has PRIVATE accessibility, the module procedure must have PRIVATE accessibility. If the module has a generic identifier, it must also be declared PRIVATE.

If a procedure has a generic identifier, the accessibility of the procedure's specific name is independent of the accessibility of its generic identifier. One can be declared PRIVATE and the other PUBLIC.

Example

The following examples show type declaration statements specifying the PUBLIC and PRIVATE attributes:

```
REAL,  PRIVATE  :: A, B, C

INTEGER, PUBLIC :: LOCAL_SUMS
```

The following is an example of the PUBLIC and PRIVATE statements:

```
MODULE SOME_DATA
  REAL ALL_B
  PUBLIC ALL_B
  TYPE RESTRICTED_DATA
    REAL LOCAL_C
    DIMENSION LOCAL_C(50)
  END TYPE RESTRICTED_DATA
  PRIVATE RESTRICTED_DATA
END MODULE
```

The following example shows a PUBLIC type with PRIVATE components:

```
MODULE MATTER
  TYPE ELEMENTS
    PRIVATE
    INTEGER C, D
  END TYPE
...
END MODULE MATTER
```

In this case, components C and D are private to type ELEMENTS, but type ELEMENTS is not private to MODULE MATTER. Any program unit that uses the module MATTER, can declare variables of type ELEMENTS, and pass as arguments values of type ELEMENTS.

The following shows another example:

```
 !LENGTH in module VECTRLEN calculates the length of a 2-D vector.
 !The module contains both private and public procedures
    MODULE VECTRLEN
      PRIVATE SQUARE
      PUBLIC LENGTH
     CONTAINS
     SUBROUTINE LENGTH(x,y,z)
        REAL,INTENT(IN) x,y
        REAL,INTENT(OUT) z
        CALL SQUARE(x,y)
        z = SQRT(x + y)
        RETURN
     END SUBROUTINE
     SUBROUTINE SQUARE(x1,y1)
        REAL x1,y1
        x1 = x1**2
        y1 = y1**2
        RETURN
     END SUBROUTINE
    END MODULE
```

See Also

- O to P
- PRIVATE
- MODULE
- TYPE
- Defining Generic Names for Procedures
- USE
- Use and Host Association

- Type Declarations
- Compatible attributes

PURE

Keyword: *Asserts that a user-defined procedure has no side effects.*

Description

This kind of procedure is specified by using the prefix PURE (or ELEMENTAL) in a FUNCTION or SUBROUTINE statement. Pure procedures are a Fortran 95 feature.

A pure procedure has no side effects. It has no effect on the state of the program, except for the following:

- For functions: It returns a value.
- For subroutines: It modifies INTENT(OUT) and INTENT(INOUT) parameters.

The following intrinsic and library procedures are implicitly pure:

- All intrinsic functions
- The elemental intrinsic subroutine MVBITS
- The intrinsic subroutine MOVE_ALLOC

A statement function is pure only if all functions that it references are pure.

Except for procedure arguments and pointer arguments, the following intent must be specified for all dummy arguments in the specification part of the procedure:

- For functions: INTENT(IN)
- For subroutines: any INTENT (IN, OUT, or INOUT)

A local variable declared in a pure procedure (including variables declared in any internal procedure) must not:

- Specify the SAVE attribute
- Be initialized in a type declaration statement or a DATA statement

The following variables have restricted use in pure procedures (and any internal procedures):

- Global variables
- Dummy arguments with INTENT(IN) (or no declared intent)

- Objects that are storage associated with any part of a global variable

They must not be used in any context that does either of the following:

- Causes their value to change. For example, they must not be used as:
  - The left side of an assignment statement or pointer assignment statement
  - An actual argument associated with a dummy argument with INTENT(OUT), INTENT(INOUT), or the POINTER attribute
  - An index variable in a DO or FORALL statement, or an implied-DO clause
  - The variable in an ASSIGN statement
  - An input item in a READ statement
  - An internal file unit in a WRITE statement
  - An object in an ALLOCATE, DEALLOCATE, or NULLIFY statement
  - An IOSTAT or SIZE specifier in an I/O statement, or the STAT specifier in a ALLOCATE or DEALLOCATE statement
- Creates a pointer to that variable. For example, they must not be used as:
  - The target in a pointer assignment statement
  - The right side of an assignment to a derived-type variable (including a pointer to a derived type) if the derived type has a pointer component at any level

A pure procedure must not contain the following:

- Any external I/O statement (including a READ or WRITE statement whose I/O unit is an external file unit number or *)
- A PAUSE statement
- A STOP statement

A pure procedure can be used in contexts where other procedures are restricted; for example:

- It can be called directly in a FORALL statement or be used in the mask expression of a FORALL statement.
- It can be called from a pure procedure. Pure procedures can only call other pure procedures.
- It can be passed as an actual argument to a pure procedure.

If a procedure is used in any of these contexts, its interface must be explicit and it must be declared pure in that interface.

Example

Consider the following:

```
PURE FUNCTION DOUBLE(X)
  REAL, INTENT(IN) :: X
  DOUBLE = 2 * X
END FUNCTION DOUBLE
```

The following shows another example:

```
PURE INTEGER FUNCTION MANDELBROT(X)
  COMPLEX, INTENT(IN) :: X
  COMPLEX__:: XTMP
  INTEGER__:: K
  ! Assume SHARED_DEFS includes the declaration
  ! INTEGER ITOL
  USE SHARED_DEFS
  K = 0
  XTMP = -X
  DO WHILE (ABS(XTMP) < 2.0 .AND. K < ITOL)
    XTMP = XTMP**2 - X
    K = K + 1
  END DO
  ITER = K
END FUNCTION
```

The following shows the preceding function used in an interface block:

```
INTERFACE

  PURE INTEGER FUNCTION MANDELBROT(X)

    COMPLEX, INTENT(IN) :: X

  END FUNCTION MANDELBROT

END INTERFACE
```

The following shows a FORALL construct calling the MANDELBROT function to update all the elements of an array:

```
FORALL (I = 1:N, J = 1:M)

  A(I,J) = MANDELBROT(COMPLX((I-1)*1.0/(N-1), (J-1)*1.0/(M-1))

END FORALL
```

See Also

- O to P
- FUNCTION
- SUBROUTINE
- FORALL
- ELEMENTAL prefix

PUTC

Portability Function: *Writes a character to Fortran external unit number 6.*

Module

USE IFPORT

Syntax

`result = PUTC (char)`

char	(Input) Character. Character to be written to external unit 6.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT
integer(4) i4
character*1 char1
do i = 1,26
  char1 = char(123-i)
  i4 = putc(char1)
  if (i4.ne.0) iflag = 1
enddo
```

See Also

- O to P
- GETC
- WRITE
- PRINT
- FPUTC

Building Applications: Portability Library Overview

PUTIMAGE, PUTIMAGE_W (W*32, W*64)

Graphics Subroutines: *Transfer the image stored in memory to the screen.*

Module

USE IFQWIN

Syntax

```
CALL PUTIMAGE (x,y,image,action)
CALL PUTIMAGE_W (wx,wy,image,action)
```

x, y	(Input) INTEGER(2). Viewport coordinates for upper-left corner of the image when placed on the screen.

wx, wy (Input) REAL(8). Window coordinates for upper-left corner of the image when placed on the screen.

image (Input) INTEGER(1). Array of single-byte integers. Stored image buffer.

action (Input) INTEGER(2). Interaction of the stored image with the existing screen image. One of the following symbolic constants (defined in `IFQWIN.F90`):

- $GAND - Forms a new screen display as the logical AND of the stored image and the existing screen display. Points that have the same color in both the existing screen image and the stored image remain the same color, while points that have different colors are joined by a logical AND.
- $GOR - Superimposes the stored image onto the existing screen display. The resulting image is the logical OR of the image.
- $GPRESET - Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by GETIMAGE, producing a negative image.
- $GPSET - Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by GETIMAGE.
- $GXOR - Causes points in the existing screen image to be inverted wherever a point exists in the stored image. This behavior is like that of a cursor. If you perform an exclusive OR of an image with the background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The $GXOR constant is a special mode often used for animation.
- In addition, the following ternary raster operation constants can be used (described in the online documentation for the Windows* API BitBlt):
  - $GSRCCOPY (same as $GPSET)
  - $GSRCPAINT (same as $GOR)
  - $GSRCAND (same as $GAND)
  - $GSRCINVERT (same as $GXOR)

- $GSRCERASE
- $GNOTSRCCOPY (same as $GPRESET)
- $GNOTSRCERASE
- $GMERGECOPY
- $GMERGEPAINT
- $GPATCOPY
- $GPATPAINT
- $GPATINVERT
- $GDSTINVERT
- $GBLACKNESS
- $GWHITENESS

PUTIMAGE places the upper-left corner of the image at the viewport coordinates (*x*, *y*).
PUTIMAGE_W places the upper-left corner of the image at the window coordinates (*wx*, *wy*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics App.
USE IFQWIN
INTEGER(1), ALLOCATABLE :: buffer(:)
INTEGER(2) status, x
INTEGER(4) imsize
status = SETCOLOR(INT2(4))
! draw a circle
status = ELLIPSE($GFILLINTERIOR,INT2(40),INT2(55),    &
                 INT2(70),INT2(85))
imsize = IMAGESIZE (INT2(39),INT2(54),INT2(71), &
                 INT2(86))
ALLOCATE (buffer(imsize))
CALL GETIMAGE(INT2(39),INT2(54),INT2(71),INT2(86),    &
             buffer)
! copy a row of circles beneath it
DO x = 5 , 395, 35
   CALL PUTIMAGE(x, INT2(90), buffer, $GPSET)
END DO
DEALLOCATE(buffer)
END
```

See Also

- O to P
- GETIMAGE
- GRSTATUS
- IMAGESIZE

PXF(type)GET

POSIX Subroutine: *Gets the value stored in a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

CALL PXF(type)GET (*jhandle*,*compname*,*value*,*ierror*)

CALL PXF(type)GET (*jhandle*,*compname*,*value*,*ilen*,*ierror*) ! syntax when (type) is STR

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFINTGET
REAL	REAL(4)	PXFREALGET
LGCL	LOGICAL(4)	PXFLGCLGET
STR	CHARACTER*(*)	PXFSTRGET
CHAR	CHARACTER(1)	PXFCHARGET
DBL	REAL(8)	PXFDBLGET
INT8	INTEGER(8)	PXFINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to retrieve data from.

value (Output) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). Stores the value of the component (or field).

ilen	(Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFSTRGET). Stores the length of the returned string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXF(type)GET subroutines retrieve the value from component (or field) *compname* of the structure associated with handle *jhandle* into variable *value*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFTIMES (which shows PXFINTGET and PXFINT8GET)

See Also

- O to P
- PXF(type)SET

PXF(type)SET

POSIX Subroutine: *Sets the value of a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXF(type)SET (jhandle,compname,value,ierror)
```

```
CALL PXF(type)SET (jhandle,compname,value,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFINTSET
REAL	REAL(4)	PXFREALSET

Value	Data Type	Routine Name
LGCL	LOGICAL(4)	PXFLGCLSET
STR	CHARACTER*(*)	PXFSTRSET
CHAR	CHARACTER(1)	PXFCHARSET
DBL	REAL(8)	PXFDBLSET
INT8	INTEGER(8)	PXFINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

value (Input) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). The value for the component (or field).

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFSTRSET). The length of the string in *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFSTRUCTCREATE (which shows PXFSTRSET)

See Also

- O to P
- PXF(type)GET

PXFA<TYPE>GET

POSIX Subroutine: *Gets the array values stored in a component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFA(type)GET (jhandle,compname,value,ialen,ierror)
```

```
CALL PXFA(type)GET (jhandle,compname,value,ialen,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFAINTGET
REAL	REAL(4)	PXFAREALGET
LGCL	LOGICAL(4)	PXFALGCLGET
STR	CHARACTER*(*)	PXFASTRGET
CHAR	CHARACTER(1)	PXFACHARGET
DBL	REAL(8)	PXFADBLGET
INT8	INTEGER(8)	PXFAINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to retrieve data from.

value (Output) An array, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type of the array is INTEGER(4). Stores the value of the component (or field).

ialen (Input) INTEGER(4). The size of array *value*.

ilen	(Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFASTRGET). An array that stores the lengths of elements of array *value*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFA(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFA(type)GET subroutines are used, the entire array is accessed (read from the component or field) as a unit.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFA(type)SET
- PXF(type)GET

PXFA(type)SET

POSIX Subroutine: *Sets the value of an array component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFA(type)SET (jhandle, compname,value,ialen,ierror)
```

```
CALL PXFA(type)SET (jhandle,compname,value,ialen,ilen,ierror) ! syntax when (type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFAINTSET
REAL	REAL(4)	PXFAREALSET

Value	Data Type	Routine Name
LGCL	LOGICAL(4)	PXFALGCLSET
STR	CHARACTER*(*)	PXFASTRSET
CHAR	CHARACTER(1)	PXFACHARSET
DBL	REAL(8)	PXFADBLSET
INT8	INTEGER(8)	PXFAINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

value (Input) An array, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type of the array is INTEGER(4). The value for the component (or field).

ialen (Input) INTEGER(4). The size of array *value*.

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFASTRSET). An array that specifies the lengths of elements of array *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFA(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFA(type)GET subroutines are used, the entire array is accessed (read from the component or field) as a unit.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFA(type)GET

- PXF(type)SET

PXFACCESS

POSIX Subroutine: *Determines the accessibility of a file.*

Module

USE IFPOSIX

Syntax

CALL PXFACCESS (*path*,*ilen*,*iamode*,*ierror*)

path	(Input) Character. The name of the file.
ilen	(Input) INTEGER(4). The length of the *path* string.
iamode	(Input) INTEGER(4). One or more of the following:

0	Checks for existence of the file.
1[1]	Checks for execute permission.
2	Checks for write access.
4	Checks for read access.
6	Checks for read/write access.

[1]L*X only

ierror	(Output) INTEGER(4). The error status.

If access is permitted, the result value is zero; otherwise, an error code. Possible error codes are:

- -1: A bad parameter was passed.
- ENOENT: The named directory does not exist.
- EACCES: Access requested was denied.

On Windows* systems, if the name given is a directory name, the function only checks for existence. All directories have read/write access on Windows* systems.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFALARM

POSIX Subroutine: *Schedules an alarm.*

Module

USE IFPOSIX

Syntax

CALL PXFALARM (*iseconds*,*isecleft*,*ierror*)

iseconds	(Input) INTEGER(4). The number of seconds before the alarm signal should be delivered.
isecleft	(Output) INTEGER(4). The number of seconds remaining until any previously scheduled alarm signal is due to be delivered. It is set to zero if there was no previously scheduled alarm signal.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFALARM subroutine arranges for a SIGALRM signal to be delivered to the process in seconds *iseconds*.

On Linux* and Mac OS* X systems, SIGALRM is a reserved defined constant that is equal to 14. You can use any other routine to install the signal handler. You can get SIGALRM and other signal values by using PXFCONST or IPXFCONST.

On Windows* systems, the SIGALRM feature is not supported, but the POSIX library has an implementation you can use. You can provide a signal handler for SIGALRM by using PXFSIGACTION.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFCONST
- IPXFCONST

- PXFSIGACTION

PXFCALLSUBHANDLE

POSIX Subroutine: *Calls the associated subroutine.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCALLSUBHANDLE (jhandle2,ival,ierror)
```

jhandle2	(Input) INTEGER(4). A handle to the subroutine.
ival	(Input) INTEGER(4). The argument to the subroutine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFCALLSUBHANDLE subroutine, when given a subroutine handle, calls the associated subroutine.

PXFGETSUBHANDLE should be used to obtain a subroutine handle.

NOTE. The subroutine cannot be a function, an intrinsic, or an entry point, and must be defined with exactly one integer argument.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFGETSUBHANDLE

PXFCFGETISPEED (L*X, M*X)

POSIX Subroutine: *Returns the input baud rate from a* `termios` *structure.*

Module

USE IFPOSIX

Syntax

`CALL PXFCFGETISPEED (jtermios,iospeed,ierror)`

jtermios	(Input) INTEGER(4). A handle of structure `termios`.
iospeed	(Output) INTEGER(4). The returned value of the input baud rate from the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFCFSETISPEED

PXFCFGETOSPEED (L*X, M*X)

POSIX Subroutine: *Returns the output baud rate from a* `termios` *structure.*

Module

USE IFPOSIX

Syntax

`CALL PXFCFGETOSPEED (jtermios,iospeed,ierror)`

jtermios	(Input) INTEGER(4). A handle of structure `termios`.

iospeed	(Output) INTEGER(4). The returned value of the output baud rate from the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. To get a handle for an instance of the termios structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFCFSETOSPEED

PXFCFSETISPEED (L*X, M*X)

POSIX Subroutine: *Sets the input baud rate in a termios structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCFSETISPEED (jtermios,ispeed,ierror)
```

jtermios	(Input) INTEGER(4). A handle of structure termios.
ispeed	(Input) INTEGER(4). The value of the input baud rate for the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. To get a handle for an instance of the termios structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFCFGETISPEED

PXFCFSETOSPEED (L*X, M*X)

POSIX Subroutine: *Sets the output baud rate in a* `termios` *structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCFSETOSPEED (jtermios,ispeed,ierror)
```

jtermios	(Input) INTEGER(4). A handle of structure `termios`.
ispeed	(Input) INTEGER(4). The value of the output baud rate for the structure associated with handle *jtermios*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. To get a handle for an instance of the `termios` structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFCFGETOSPEED

PXFCHDIR

POSIX Subroutine: *Changes the current working directory.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCHDIR (path,ilen,ierror)
```

path	(Input) Character. The directory to be changed to.
ilen	(Input) INTEGER(4). The length of the *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFMKDIR

PXFCHMOD

POSIX Subroutine: *Changes the ownership mode of the file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCHMOD (path,ilen,imode,ierror)
```

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of the *path* string.
imode	(Input) INTEGER(4). The ownership mode of the file. On Windows* systems, see your Microsoft* Visual C++ Installation in the `\include` directory under `sys\stat.h` for the values of *imode*. On Linux* and Mac OS* X systems, use octal file-access mode.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. On Linux and Mac OS X systems, you must have sufficient ownership permissions, such as being the owner of the file or having read/write access of the file.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFCHOWN (L*X, M*X)

POSIX Subroutine: *Changes the owner and group of a file.*

Module

USE IFPOSIX

Syntax

`CALL PXFCHOWN (path,ilen,iowner,igroup,ierror)`

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of the *path* string.
iowner	(Input) INTEGER(4). The owner UID.
igroup	(Input) INTEGER(4). The group GID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFCLEARENV

POSIX Subroutine: *Clears the process environment.*

Module

USE IFPOSIX

Syntax

`CALL PXFCLEARENV (ierror)`

ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

After a call to PXFCLEARENV, no environment variables are defined.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFCLOSE

POSIX Subroutine: *Closes the file associated with the descriptor.*

Module

USE IFPOSIX

Syntax

`CALL PXFCLOSE (fd,ierror)`

fd	(Input) INTEGER(4). A file descriptor.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFOPEN

PXFCLOSEDIR

POSIX Subroutine: *Closes the directory stream.*

Module

USE IFPOSIX

Syntax

`CALL PXFCLOSEDIR (idirid,ierror)`

idirid	(Input) INTEGER(4). The directory ID obtained from PXFOPENDIR.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFCLOSEDIR subroutine closes the directory associated with *idirid*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFOPENDIR

PXFCONST

POSIX Subroutine: *Returns the value associated with a constant.*

Module

USE IFPOSIX

Syntax

CALL PXFCONST (*constname*,*ival*,*ierror*)

constname (Input) Character. The name of one of the following constants:

- STDIN_UNIT
- STDOUT_UNIT
- STDERR_UNIT
- EINVAL
- ENONAME
- ENOHANDLE
- EARRAYLEN
- ENOENT
- ENOTDIR
- EACCES

The constants beginning with E signify various error values for the system variable errno.

ival	(Output) INTEGER(4). The returned value of the constant.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, it is set to -1.

For more information on these constants, see your Microsoft* Visual C++ documentation (Windows* systems) or the errno.h file (Linux* and Mac OS* X systems).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFISCONST

PXFCREAT

POSIX Subroutine: *Creates a new file or rewrites an existing file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFCREAT (path,ilen,imode,ifildes,ierror)
```

path	(Input) Character. The pathname of the file.
ilen	(Input) INTEGER(4). The length of *path* string.
imode	(Input) INTEGER(4). The mode of the newly created file. On Windows* systems, see your Microsoft* Visual C++ documentation for permitted mode values. On Linux* and Mac OS* X systems, use octal file-access mode.
ifildes	(Output) INTEGER(4). The returned file descriptor.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFCTERMID (L*X, M*X)

POSIX Subroutine: *Generates a terminal pathname.*

Module

USE IFPOSIX

Syntax

`CALL PXFCTERMID (s,ilen,ierror)`

s	(Output) Character. The returned pathname of the terminal.
ilen	(Output) INTEGER(4). The length of the returned value in the *s* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror*is set to zero; otherwise, an error code.

This subroutine returns a string that refers to the current controlling terminal for the current process.

PXFDUP, PXFDUP2

POSIX Subroutine: *Duplicates an existing file descriptor.*

Module

USE IFPOSIX

Syntax

`CALL PXFDUP (ifildes,ifid,ierror)`

`CALL PXFDUP2 (ifildes,ifildes2,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor to duplicate.
ifid	(Output) INTEGER(4). The returned new duplicated file descriptor.
ifildes2	(Input) INTEGER(4). The number for the new file descriptor.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFDUP subroutine creates a second file descriptor for an opened file.

The PXFDUP2 subroutine copies the file descriptor associated with *ifildes*. Integer number *ifildes2* becomes associated with this new file descriptor, but the value of *ifildes2* is not changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFE(type)GET

POSIX Subroutine: *Gets the value stored in an array element component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFE(type)GET (jhandle, compname,index,value,ierror)

CALL PXFE(type)GET  (jhandle,compname,index,value,ilen,ierror) ! syntax when
(type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFEINTGET
REAL	REAL(4)	PXFEREALGET
LGCL	LOGICAL(4)	PXFELGCLGET
STR	CHARACTER*(*)	PXFESTRGET
CHAR	CHARACTER(1)	PXFECHARGET
DBL	REAL(8)	PXFEDBLGET
INT8	INTEGER(8)	PXFEINT8GET

jhandle (Input) INTEGER(4). A handle of a structure.

compname	(Input) Character. The name of the component (or field) of the structure to retrieve data from.
index	(Input) INTEGER(4). The index of the array element to get data for.
value	(Output) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). Stores the value of the component (or field).
ilen	(Output) INTEGER(4). This argument can only be used when *(type)* is STR (PXFESTRGET). Stores the length of the returned string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFE(type)GET subroutines are similar to the PXF(type)GET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFE(type)GET subroutines are used, the array element with index *index* is accessed (read from the component or field).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFE(type)SET
- PXF(type)GET

PXFE(type)SET

POSIX Subroutine: *Sets the value of an array element component (or field) of a structure.*

Module

USE IFPOSIX

Syntax

```
CALL PXFE(type)SET (jhandle, compname,index,value,ierror)
```

```
CALL PXFE(type)SET  (jhandle,compname,index,value,ilen,ierror) ! syntax when
(type) is STR
```

(type) A placeholder for one of the following values:

Value	Data Type	Routine Name
INT	INTEGER(4)	PXFEINTSET
REAL	REAL(4)	PXFEREALSET
LGCL	LOGICAL(4)	PXFELGCLSET
STR	CHARACTER*(*)	PXFESTRSET
CHAR	CHARACTER(1)	PXFECHARSET
DBL	REAL(8)	PXFEDBLSET
INT8	INTEGER(8)	PXFEINT8SET

jhandle (Input) INTEGER(4). A handle of a structure.

compname (Input) Character. The name of the component (or field) of the structure to write data to.

index (Input) INTEGER(4). The index of the array element to write data to.

value (Input) A variable, whose data type depends on the value of *(type)*. See the table above for the data types for each value; for example, if the value for *(type)* is INT, the data type is INTEGER(4). The value for the component (or field).

ilen (Input) INTEGER(4). This argument can only be used when *(type)* is STR (PXFESTRSET). The length of the string *value*.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFE(type)SET subroutines are similar to the PXF(type)SET subroutines, but they should be used when the component (or field) of the structure is an array.

When the PXFE(type)SET subroutines are used, the array element with index *index* is accessed (written to the component or field).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFE(type)GET
- PXF(type)SET

PXFEXECV

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

CALL PXFEXECV (*path*,*lenpath*,*argv*,*lenargv*,*iargc*,*ierror*)

path	(Input) Character. The path to the new executable process.
lenpath	(Input) INTEGER(4). The length of *path* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECV subroutine executes a new executable process (file) by passing command-line arguments specified in the argv array. If execution is successful, no return is made to the calling process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFEXECVE
- PXFEXECVP

PXFEXECVE

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

CALL PXFEXECVE (*path*,*lenpath*,*argv*,*lenargv*,*iargc*,*env*,*lenenv*,*ienvc*,*ierror*)

path	(Input) Character. The path to the new executable process.
lenpath	(Input) INTEGER(4). The length of *path* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments.
env	(Input) An array of character strings. Contains the environment settings for the new process.
lenenv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *env*.
ienvc	(Input) INTEGER(4). The number of environment settings in *env*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECVE subroutine executes a new executable process (file) by passing command-line arguments specified in the *argv* array and environment settings specified in the *env* array.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFEXECV
- PXFEXECVP

PXFEXECVP

POSIX Subroutine: *Executes a new process by passing command-line arguments.*

Module

USE IFPOSIX

Syntax

CALL PXFEXECVP (*file*,*lenfile*,*argv*,*lenargv*,*iargc*,*ierror*)

file	(Input) Character. The filename of the new executable process.
lenfile	(Input) INTEGER(4). The length of *file* string.
argv	(Input) An array of character strings. Contains the command-line arguments to be passed to the new process.
lenargv	(Input) INTEGER(4). An array that contains the lengths for each corresponding character string in *argv*.
iargc	(Input) INTEGER(4). The number of command-line arguments.
ierror	(Input) Character. The filename of the new executable process.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFEXECVP subroutine executes a new executable process(file) by passing command-line arguments specified in the*argv* array. It uses the PATH environment variable to find the file to execute.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFEXECV
- PXFEXECVE

PXFEXIT, PXFFASTEXIT

POSIX Subroutine: *Exits from a process.*

Module

USE IFPOSIX

Syntax

`CALL PXFEXIT (istatus)`

`CALL PXFFASTEXIT (istatus)`

istatus (Input) INTEGER(4). The exit value.

The PXFEXIT subroutine terminates the calling process. It calls, in last-in-first-out (LIFO) order, the functions registered by C runtime functions `atexit` and `onexit`, and flushes all file buffers before terminating the process. The *istatus* value is typically set to zero to indicate a normal exit and some other value to indicate an error.

The PXFFASTEXIT subroutine terminates the calling process without processing `atexit` or `onexit`, and without flushing stream buffers.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program t1

use ifposix

integer(4) ipid, istat, ierror, ipid_ret, istat_ret

  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
```

```
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if

  end if
10 FORMAT (A,Z)

end program
```

PXFFCNTL (L*X, M*X)

POSIX Subroutine: *Manipulates an open file descriptor.*

Module

USE IFPOSIX

Syntax

CALL PXFFCNTL (*ifildes*,*icmd*,*iargin*,*iargout*,*ierror*)

ifildes	(Input) INTEGER(4). A file descriptor.
icmd	(Input) INTEGER(4). Defines an action for the file descriptor.
iargin	(Input; output) INTEGER(4). Interpretation of this argument depends on the value of *icmd*.
iargout	(Output) INTEGER(4). Interpretation of this argument depends on the value of *icmd*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFFCNTL is a multi-purpose subroutine that causes an action to be performed on a file descriptor. The action, defined in *icmd*, can be obtained by using the values of predefined macros in C header fcntl.h, or by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
F_DUPFD	Returns into *iargout* the lowest available unopened file descriptor greater than or equal to *iargin*. The new file descriptor refers to the same open file as *ifildes* and shares any locks. The system flag FD_CLOEXEC for the new file descriptor is cleared so the new descriptor will not be closed on a call to PXFEXEC subroutine.
F_GETFD	Returns into *iargout* the value of system flag FD_CLOEXEC associated with *ifildes*. In this case, *iargin* is ignored.
F_SETFD	Sets or clears the system flag FD_CLOEXEC for file descriptor *ifildes*. The PXFEXEC family of functions will close all file descriptors with the FD_CLOEXEC flag set. The value for FD_CLOEXEC is obtained from argument *iargin*.

Constant	Action
F_GETFL	Returns the file status flags for file descriptor *ifildes*. Unlike F_GETFD, these flags are associated with the file and shared by all descriptors. A combination of the following flags, which are symbolic names for PXFCONST or IPXFCONST, can be returned: • O_APPEND - Specifies the file is opened in append mode. • O_NONBLOCK - Specifies when the file is opened, it does not block waiting for data to become available. • O_RDONLY - Specifies the file is opened for reading only. • O_RDWR - Specifies the file is opened for both reading and writing. • O_WRONLY - Specifies the file is opened for writing only.
F_SETFL	Sets the file status flags from *iargin* for file descriptor *ifildes*. Only O_APPEND or O_NONBLOCK flags can be modified. In this case, *iargout* is ignored.
F_GETLK	Gets information about a lock. Argument *iargin* must be a handle of structure `flock`. This structure is taken as the description of a lock for the file. If there is a lock already in place that would prevent this lock from being locked, it is returned to the structure associated with handle *iargin*. If there are no locks in place that would prevent the lock from being locked, field l_type in the structure is set to the value of the constant with symbolic name F_UNLCK.

Constant	Action
F_SETLK	Sets or clears a lock. Argument *iargin* must be a handle of structure `flock`. The lock is set or cleared according to the value of structure field l_type. If the lock is busy, an error is returned.
F_SETLKW	Sets or clears a lock, but causes the process to wait if the lock is busy. Argument *iargin* must be a handle of structure `flock`. The lock is set or cleared according to the value of structure field l_type. If the lock is busy, PXFCNTL waits for an unlock.

NOTE. To get a handle for an instance of the `flock` structure, use PXFSTRUCTCREATE with the string 'flock' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- IPXFCONST
- PXFCONST

PXFFDOPEN

POSIX Subroutine: *Opens an external unit.*

Module

USE IFPOSIX

Syntax

`CALL PXFFDOPEN (ifildes,iunit,access,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor of the opened file.
iunit	(Input) INTEGER(4). The Fortran logical unit to connect to file descriptor *ifildes*.

access (Input) Character. A character string that specifies the attributes for the Fortran unit. The string must consist of one or more of the following keyword/value pairs. Keyword/value pairs should be separated by a comma, and blanks are ignored.

Keyword	Possible Values	Description	Default
'NEWLINE'	'YES' or 'NO'	I/O type	'YES'
'BLANK'	'NULL' or 'ZERO'	Interpretation of blanks	'NULL'
'STATUS'	'OLD', 'SCRATCH', or 'UNKNOWN'	File status at open	'UNKNOWN'
'FORM'	'FORMATTED' or 'UNFORMATTED'	Format type	'FORMATTED'

Keywords should be separated from their values by the equals ('=') character; for example:

```
call PXFDOPEN (IFILDES, IUNIT, 'BLANK=NULL, STATUS=UNKNOWN',
IERROR)
```

ierror (Output) INTEGER(4). The error status.

The PXFFDOPEN subroutine connects an external unit identified by *iunit* to a file descriptor *ifildes*. If unit is already connected to a file, the file should be closed before using PXFFDOPEN.

NOTE. On Windows* systems, the default value of the POSIX/IO flag is 0, which causes PXFFDOPEN to return an error.

To prevent this, call subroutine PXFPOSIXIO and set the value of the POSIX/IO flag to 1.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFPOSIXIO

PXFFFLUSH

POSIX Subroutine: *Flushes a file directly to disk.*

Module

USE IFPOSIX

Syntax

`CALL PXFFFLUSH (lunit,ierror)`

lunit	(Input) INTEGER(4). A Fortran logical unit.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFFLUSH subroutine writes any buffered output to the file connected to unit *lunit*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFFGETC

POSIX Subroutine: *Reads a character from a file.*

Module

USE IFPOSIX

Syntax

`CALL PXFFGETC (lunit,char,ierror)`

lunit	(Input) INTEGER(4). A Fortran logical unit.
char	(Input) Character. The character to be read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFGETC subroutine reads a character from a file connected to unit *lunit*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFFPUTC

PXFFILENO

POSIX Subroutine: *Returns the file descriptor associated with a specified unit.*

Module

USE IFPOSIX

Syntax

`CALL PXFFILENO (lunit,fd,ierror)`

lunit	(Input) INTEGER(4). A Fortran logical unit.
fd	(Output) INTEGER(4). The returned file descriptor.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. Possible error codes are:

- EINVAL: *lunit* is not an open unit.
- EBADF: *lunit* is not connected with a file descriptor.

The PXFFILENO subroutine returns in *fd* the file descriptor associated with *lunit*.

NOTE. On Windows* systems, the default value of the POSIX/IO flag is 0, which prevents OPEN from connecting a unit to a file descriptor and causes PXFFILENO to return an error.

To prevent this, call subroutine PXFPOSIXIO and set the value of the POSIX/IO flag to 1. This setting allows a connection to a file descriptor.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFPOSIXIO

PXFFORK (L*X, M*X)

POSIX Subroutine: *Creates a child process that differs from the parent process only in its PID.*

Module

USE IFPOSIX

Syntax

CALL PXFFORK (*ipid*,*ierror*)

ipid	(Output) INTEGER(4). The returned PID of the new child process.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFORK subroutine creates a child process that differs from the parent process only in its PID. If successful, the PID of the child process is returned in the parent's thread of execution, and a zero is returned in the child's thread of execution. Otherwise, a -1 is returned in the parent's context and no child process is created.

Example

```
program t1

use ifposix

integer(4) ipid, istat, ierror, ipid_ret, istat_ret

  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
```

```
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if

  end if
10 FORMAT (A,Z)

end program
```

See Also

- O to P
- IPXWEXITSTATUS

PXFFPATHCONF

POSIX Subroutine: *Gets the value for a configuration option of an opened file.*

Module

USE IFPOSIX

Syntax

CALL PXFFPATHCONF (*ifildes*,*name*,*ival*,*ierror*)

ifildes	(Input) INTEGER(4). The file descriptor of the opened file.
name	(Input) INTEGER(4). The configurable option.
ival	(Output) INTEGER(4). The value of the configurable option.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFPATHCONF subroutine gets a value for the configuration option named for the opened file with descriptor *ifildes*.

The configuration option, defined in *name*, can be obtained by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
_PC_LINK_MAX	Returns the maximum number of links to the file. If *ifildes* refers to a directory, then the value applies to the whole directory.
_PC_MAX_CANON[1]	Returns the maximum length of a formatted input line; the file descriptor *ifildes* must refer to a terminal.
_PC_MAX_INPUT[1]	Returns the maximum length of an input line; the file descriptor *ifildes* must refer to a terminal.
_PC_NAME_MAX	Returns the maximum length of a filename in *ifildes* that the process is allowed to create.

Constant	Action
_PC_PATH_MAX	Returns the maximum length of a relative pathname when *ifildes* is the current working directory.
_PC_PIPE_BUF	Returns the size of the pipe buffer; the file descriptor *ifildes* must refer to a pipe or FIFO.
_PC_CHOWN_RESTRICTED[1]	Returns nonzero if PXFCHOWN may not be used on this file. If *ifildes* refers to a directory, then this applies to all files in that directory.
_PC_NO_TRUNC[1]	Returns nonzero if accessing filenames longer than _POSIX_NAME_MAX will generate an error.
_PC_VDISABLE[1]	Returns nonzero if special character processing can be disabled; the file descriptor *ifildes* must refer to a terminal.

[1]L*X, M*X

On Linux* and Mac OS* X systems, the corresponding macros are defined in <unistd.h>. The values for *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in <unistd.h>. The following table shows the corresponding macro names for the above constants:

Constant	Corresponding Macro
_PC_LINK_MAX	_POSIX_LINK_MAX
_PC_MAX_CANON	_POSIX_MAX_CANON
_PC_MAX_INPUT	_POSIX_MAX_INPUT
_PC_NAME_MAX	_POSIX_NAME_MAX
_PC_PATH_MAX	_POSIX_PATH_MAX

Constant	Corresponding Macro
_PC_PIPE_BUF	_POSIX_PIPE_BUF
_PC_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED
_PC_NO_TRUNC	_POSIX_NO_TRUNC
_PC_VDISABLE	_POSIX_VDISABLE

See Also

- O to P
- IPXFCONST
- PXFCONST
- PXFPATHCONF

PXFFPUTC

POSIX Subroutine: *Writes a character to a file.*

Module

USE IFPOSIX

Syntax

CALL PXFFPUTC (*lunit*,*char*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
char	(Input) Character. The character to be written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EEND if the end of the file has been reached.

The PXFFPUTC subroutine writes a character to the file connected to unit *lunit*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFFGETC

PXFFSEEK

POSIX Subroutine: *Modifies a file position.*

Module

USE IFPOSIX

Syntax

`CALL PXFFSEEK (lunit,ioffset,iwhence,ierror)`

lunit	(Input) INTEGER(4). A Fortran logical unit.
ioffset	(Input) INTEGER(4). The number of bytes away from *iwhence* to place the pointer.
iwhence	(Input) INTEGER(4). The position within the file. The value must be one of the following constants (defined in `stdio.h`):

SEEK_SET = 0	Offset from the beginning of the file.
SEEK_CUR = 1	Offset from the current position of the file pointer.
SEEK_END = 2	Offset from the end of the file.

ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. Possible error codes are:

- EINVAL: No file is connected to *lunit*, *iwhence* is not a proper value, or the resulting offset is invalid.
- ESPIPE: *lunit* is a pipe or FIFO.
- EEND: The end of the file has been reached.

The PXFFSEEK subroutine modifies the position of the file connected to unit *lunit*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFFSTAT

POSIX Subroutine: *Gets a file's status information.*

Module

USE IFPOSIX

Syntax

`CALL PXFFSTAT (ifildes,jstat,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor for an opened file.
jstat	(Input) INTEGER(4). A handle of structure `stat`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFFSTAT subroutine puts the status information for the file associated with *ifildes* into the structure associated with handle *jstat*.

NOTE. To get a handle for an instance of the `stat` structure, use PXFSTRUCTCREATE with the string 'stat' for the structure name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFSTRUCTCREATE

PXFFTELL

POSIX Subroutine: *Returns the relative position in bytes from the beginning of the file.*

Module

USE IFPOSIX

Syntax

CALL PXFFTELL (*lunit*,*ioffset*,*ierror*)

lunit	(Input) INTEGER(4). A Fortran logical unit.
ioffset	(Output) INTEGER(4). The returned relative position in bytes from the beginning of the file.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFGETARG

POSIX Subroutine: *Gets the specified command-line argument.*

Module

USE IFPOSIX

Syntax

CALL PXFGETARG (*argnum*,*str*,*istr*,*ierror*)

argnum	(Input) INTEGER(4). The number of the command-line argument.
str	(Output) Character. The returned string value.
istr	(Output) INTEGER(4). The length of the returned string; it is zero if an error occurs.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETARG subroutine places the command-line argument with number *argnum* into character string *str*. If *argnum* is equal to zero, the value of the argument returned is the command name of the executable file.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- IPXFARGC

PXFGETATTY

POSIX Subroutine: *Tests whether a file descriptor is connected to a terminal.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETATTY (ifildes,isatty,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor.
isatty	(Output) LOGICAL(4). The returned value.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If file descriptor *ifildes* is open and connected to a terminal, *isatty* returns .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFGETC

POSIX Subroutine: *Reads a character from standard input unit 5.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETC (nextcar,ierror)`

nextcar	(Output) Character. The returned character that was read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFPUTC

PXFGETCWD

POSIX Subroutine: *Returns the path of the current working directory.*

Module

USE IFPOSIX

Syntax

CALL PXFGETCWD (*buf*,*ilen*,*ierror*)

buf	(Output) Character. The returned pathname of the current working directory.
ilen	(Output) INTEGER(4). The length of the returned pathname.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EINVAL if the size of *buf* is insufficient.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFGETEGID (L*X, M*X)

POSIX Subroutine: *Gets the effective group ID of the current process.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETEGID (iegid,ierror)
```

iegid	(Output) INTEGER(4). The returned effective group ID.
ierror	(Output) INTEGER(4). The error status.

Description

If successful, *ierror* is set to zero; otherwise, an error code.

The effective ID corresponds to the set ID bit on the file being executed.

PXFGETENV

POSIX Subroutine: *Gets the setting of an environment variable.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETENV (name,lenname,value,lenval,ierror)
```

name	(Input) Character. The name of the environment variable.
lenname	(Input) INTEGER(4). The length of *name*.
value	(Output) Character. The returned value of the environment variable.
lenval	(Output) INTEGER(4). The returned length of *value*. If an error occurs, it returns zero.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFSETENV

PXFGETEUID (L*X, M*X)

POSIX Subroutine: *Gets the effective user ID of the current process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETEUID (ieuid,ierror)`

ieuid	(Output) INTEGER(4). The returned effective user ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The effective ID corresponds to the set ID bit on the file being executed.

PXFGETGID (L*X, M*X)

POSIX Subroutine: *Gets the real group ID of the current process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETGID (igid,ierror)`

igid	(Output) INTEGER(4). The returned real group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The real ID corresponds to the ID of the calling process.

Example

See the example in PXFGETGROUPS

See Also

- O to P

- PXFSETGID

PXFGETGRGID (L*X, M*X)

POSIX Subroutine: *Gets group information for the specified GID.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETGRGID (jgid,jgroup,ierror)`

jgid	(Input) INTEGER(4). The group ID to retrieve information about.
jgroup	(Input) INTEGER(4). A handle of structure `group`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETGRGID subroutine stores the group information from `/etc/group` for the entry that matches the group GID *jgid* in the structure associated with handle *jgroup*.

NOTE. To get a handle for an instance of the `group`structure, use PXFSTRUCTCREATE with the string 'group' for the structure name.

Example

See the example in PXFGETGROUPS

See Also

- O to P
- PXFSTRUCTCREATE

PXFGETGRNAM (L*X, M*X)

POSIX Subroutine: *Gets group information for the named group.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETGRNAM (name,ilen,jgroup,ierror)
```

name	(Input) Character. The name of the group to retrieve information about.
ilen	(Input) INTEGER(4). The length of the *name* string.
jgroup	(Input) INTEGER(4). A handle of structure group.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETGRNAM subroutine stores the group information from `/etc/group` for the entry that matches the group name name in the structure associated with handle *jgroup*.

NOTE. To get a handle for an instance of the `group` structure, use PXFSTRUCTCREATE with the string 'group' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE

PXFGETGROUPS (L*X, M*X)

POSIX Subroutine: *Gets supplementary group IDs.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETGROUPS (igidsetsize,igrouplist,ngroups,ierror)
```

igidsetsize	(Input) INTEGER(4). The number of elements in the *igrouplist* array.
igrouplist	(Output) INTEGER(4). The array that has the returned supplementary group IDs.
ngroups	(Output) INTEGER(4). The total number of supplementary group IDs for the process.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETGROUPS subroutine returns, up to size *igidsetsize*, the supplementary group IDs in array *igrouplist*. It is unspecified whether the effective group ID of the calling process is included in the returned list. If the size is zero, the list is not modified, but the total number of supplementary group IDs for the process is returned.

Example

```
program test5
  use ifposix
  implicit none
  integer(4) number_of_groups, ierror, isize,i, igid
  integer(4),allocatable,dimension(:):: igrouplist
  integer(JHANDLE_SIZE) jgroup
  ! Get total number of groups in system
  ! call PXFGETGROUPS with 0

  call PXFGETGROUPS(0, igrouplist, number_of_groups, ierror)
  if(ierror.NE.0) STOP 'Error: first call of PXFGETGROUPS fails'
  print *," The number of groups in system ", number_of_groups
  ! Get Group IDs
  isize = number_of_groups
  ALLOCATE( igrouplist(isize))
  call PXFGETGROUPS(isize, igrouplist, number_of_groups, ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     STOP 'Error: first call of PXFGETGROUPS fails'
  end if

  print *," Create an instance for structure 'group' "
```

```
call PXFSTRUCTCREATE("group",jgroup, ierror)
if(ierror.NE.0) then
   DEALLOCATE(igrouplist)
   STOP 'Error: PXFSTRUCTCREATE failed to create an instance of group'
end if

do i=1,  number_of_groups
   call PXFGETGRGID( igrouplist(i), jgroup, ierror)
   if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     call PXFSTRUCTFREE(jgroup, ierror)
     print *,'Error: PXFGETGRGID failed for i=',i," gid=", igrouplist(i)
     STOP 'Abnormal termination'
   end if

   call PRINT_GROUP_INFO(jgroup)

end do

call PXFGETGID(igid,ierror)
if(ierror.NE.0) then
   DEALLOCATE(igrouplist)
   call PXFSTRUCTFREE(jgroup, ierror)
   print *,'Error: PXFGETGID failed'

   STOP 'Abnormal termination'
end if
```

```
  call PXFGETGRGID( igid, jgroup, ierror)
  if(ierror.NE.0) then
     DEALLOCATE(igrouplist)
     call PXFSTRUCTFREE(jgroup, ierror)
     print *,"Error: PXFGETGRGID failed for  gid=", igid
     STOP 'Abnormal termination'
  end if

  call PRINT_GROUP_INFO(jgroup)

  DEALLOCATE(igrouplist)
  call PXFSTRUCTFREE(jgroup, ierror)
  print *," Program will normal terminated"
  call PXFEXIT(0)
end
```

PXFGETLOGIN

POSIX Subroutine: *Gets the name of the user.*

Module

USE IFPOSIX

Syntax

CALL PXFGETLOGIN (*s*,*ilen*,*ierror*)

s	(Output) Character. The returned user name.
ilen	(Output) INTEGER(4). The length of the string stored in *s*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFGETPGRP (L*X, M*X)

POSIX Subroutine: *Gets the process group ID of the calling process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETPGRP (ipgrp,ierror)`

ipgrp	(Output) INTEGER(4). The returned process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Each process group is a member of a session and each process is a member of the session in which its process group is a member.

PXFGETPID

POSIX Subroutine: *Gets the process ID of the calling process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETPID (ipid,ierror)`

ipid	(Output) INTEGER(4). The returned process ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program t1
```

```
use ifposix

integer(4) ipid, istat, ierror, ipid_ret, istat_ret

  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)

  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if

  end if
10 FORMAT (A,Z)
```

```
end program
```

PXFGETPPID

POSIX Subroutine: *Gets the process ID of the parent of the calling process.*

Module

USE IFPOSIX

Syntax

CALL PXFGETPPID (*ippid*,*ierror*)

ippid	(Output) INTEGER(4). The returned process ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program t1

use ifposix

integer(4) ipid, istat, ierror, ipid_ret, istat_ret

  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
```

```
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if

  end if
10 FORMAT (A,Z)

end program
```

PXFGETPWNAM (L*X, M*X)

POSIX Subroutine: *Gets password information for a specified name.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETPWNAM (name,ilen,jpasswd,ierror)
```

name	(Input) Character. The login name of the user to retrieve information about. For example, a login name might be "jsmith", while the actual name is "John Smith".
ilen	(Input) INTEGER(4). The length of the *name* string.
jpasswd	(Input) INTEGER(4). A handle of structure *compnam*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFGETPWNAM subroutine stores the user information from /etc/passwd for the entry that matches the user name *name* in the structure associated with handle *jpasswd*.

NOTE. To get a handle for an instance of the compnam structure, use PXFSTRUCTCREATE with the string 'compnam' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE

PXFGETPWUID (L*X, M*X)

POSIX Subroutine: *Gets password information for a specified UID.*

Module

USE IFPOSIX

Syntax

```
CALL PXFGETPWUID (iuid,jpasswd,ierror)
```

iuid	(Input) INTEGER(4). The user ID to retrieve information about.
jpasswd	(Output) INTEGER(4). A handle of structure compnam.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFGETPWUID subroutine stores the user information from /etc/passwd for the entry that matches the user ID *iuid* in the structure associated with handle *jpasswd*.

NOTE. To get a handle for an instance of the compnam structure, use PXFSTRUCTCREATE with the string 'compnam' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE

PXFGETSUBHANDLE

POSIX Subroutine: *Returns a handle for a subroutine.*

Module

USE IFPOSIX

Syntax

CALL PXFGETSUBHANDLE (*sub*,*jhandle1*,*ierror*)

sub	(Input) The Fortran subroutine to get a handle for.
jhandle1	(Output) INTEGER(4). The returned handle for the subroutine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. The argument *sub* cannot be a function, an intrinsic, or an entry point, and must be defined with exactly one integer argument.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFGETUID (L*X, M*X)

POSIX Subroutine: *Gets the real user ID of the current process.*

Module

USE IFPOSIX

Syntax

`CALL PXFGETUID (iuid,ierror)`

iuid	(Output) INTEGER(4). The returned real user ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The real ID corresponds to the ID of the calling process.

See Also

- O to P
- PXFSETUID

PXFISBLK

POSIX Function: *Tests for a block special file.*

Module

USE IFPOSIX

Syntax

`result = PXFISBLK (m)`

m	(Input) INTEGER(4). The value of the `st_mode`component (field) in the structure `stat`.

Results

The result type is logical. If the file is a block special file, the result value is .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFISCHR

PXFISCHR

POSIX Function: *Tests for a character file.*

Module

USE IFPOSIX

Syntax

```
result = PXFISCHR (m)
```

m	(Input) INTEGER(4). The value of the `st_mode`component (field) in the structure `stat`.

Results

The result type is logical. If the file is a character file, the result value is .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFISBLK

PXFISCONST

POSIX Function: *Tests whether a string is a valid constant name.*

Module

USE IFPOSIX

Syntax

```
result = PXFISCONST (s)
```

s (Input) Character. The name of the constant to test.

Results

The result type is logical. The PXFISCONST function confirms whether the argument is a valid constant name that can be passed to functions PXFCONST and IPXFCONST. It returns .TRUE. only if IPXFCONST will return a valid value for name *s*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- IPXFCONST
- PXFCONST

PXFISDIR

POSIX Function: *Tests whether a file is a directory.*

Module

USE IFPOSIX

Syntax

```
result = PXFISDIR (m)
```

m (Input) INTEGER(4).The value of the `st_mode` component (field) in the structure `stat`.

Results

The result type is logical. If the file is a directory, the result value is .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFISFIFO

POSIX Function: *Tests whether a file is a special FIFO file.*

Module

USE IFPOSIX

Syntax

```
result = PXFISFIFO (m)
```

m (Input) INTEGER(4). The value of the `st_mode`component (field) in the structure `stat`.

Results

The result type is logical.

The PXFISFIFO function tests whether the file is a special FIFO file created by PXFMKFIFO. If the file is a special FIFO file, the result value is .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFMKFIFO
- PXFISREG

PXFISREG

POSIX Function: *Tests whether a file is a regular file.*

Module

USE IFPOSIX

Syntax

```
result = PXFISREG (m)
```

m	(Input) INTEGER(4). The value of the st_mode component (field) in the structure stat.

Results

The result type is logical. If the file is a regular file, the result value is .TRUE.; otherwise, .FALSE..

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFMKFIFO
- PXFISFIFO

PXFKILL

POSIX Subroutine: *Sends a signal to a specified process.*

Module

USE IFPOSIX

Syntax

CALL PXFKILL (*ipid*,*isig*,*ierror*)

ipid	(Input) INTEGER(4). The process to kill. It is determined by one of the following values:

> 0	Kills the specific process.
< 0	Kills all processes in the group.
== 0	Kills all processes in the group except special processes.
== pid_t-1	Kills all processes.

isig	(Input) INTEGER(4). The value of the signal to be sent.

ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFKILL subroutine sends a signal with value *isig* to a specified process. On Windows* systems, only the *ipid* for the current process can be used.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFLINK

POSIX Subroutine: *Creates a link to a file or directory.*

Module

USE IFPOSIX

Syntax

CALL PXFLINK (*existing*,*lenexist*,*new*,*lennew*,*ierror*)

existing	(Input) Character. The path to the file or directory you want to link to.
lenexist	(Input) INTEGER(4). The length of the *existing* string.
new	(Input) Character. The name of the new link file.
lennew	(Input) INTEGER(4). The length of the *new* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is not changed; otherwise, an error code.

The PXFLINK subroutine creates a new link (also known as a hard link) to an existing file. This new name can be used exactly as the old one for any operation. Both names refer to the same file (so they have the same permissions and ownership) and it is impossible to tell which name was the "original".

NOTE. On Windows* systems, this subroutine is only valid for NTFS file systems; for FAT systems, it returns an error.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFLOCALTIME

POSIX Subroutine: *Converts a given elapsed time in seconds to local time.*

Module

USE IFPOSIX

Syntax

`CALL PXFLOCALTIME (`*`isecnds`*`,`*`iatime`*`,`*`ierror`*`)`

isecnds (Input) INTEGER(4). The elapsed time in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970.

iatime (Output) INTEGER(4). One-dimensional array with 9 elements containing numeric time data. The elements of *iatime* are returned as follows:

Element	Value
iatime(1)	Seconds (0-59)
iatime(2)	Minutes (0-59)
iatime(3)	Hours (0-23)
iatime(4)	Day of month (1-31)
iatime(5)	Month (1-12)
iatime(6)	Gregorian year (for example, 1990)
iatime(7)	Day of week (0-6, where 0 is Sunday)
iatime(8)	Day of year (1-366)

Element	Value
iatime(9)	Daylight savings flag (1 if daylight savings time is in effect; otherwise, 0)

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFLOCALTIME subroutine converts the time (in seconds since epoch) in the *isecnds* argument to the local date and time as described by the array *iatime* above.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFLSEEK

POSIX Subroutine: *Positions a file a specified distance in bytes.*

Module

USE IFPOSIX

Syntax

CALL PXFLSEEK (*ifildes*,*ioffset*,*iwhence*,*iposition*,*ierror*)

ifildes (Input) INTEGER(4). A file descriptor.

ioffset (Input) INTEGER(4). The number of bytes to move.

iwhence (Input) INTEGER(4). The starting position. The value must be one of the following:

- SEEK_SET = 0

  Sets the offset to *ioffset* bytes.
- SEEK_CUR = 1

  Sets the offset to its current location plus *ioffset* bytes.
- SEEK_END = 2

  Sets the offset to the size of the file plus *ioffset* bytes.

iposition	(Output) INTEGER(4). The ending position; the resulting offset location as measured in bytes from the beginning of the file.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFLSEEK subroutine repositions the offset of file descriptor *ifildes* to the argument *ioffset* according to the value of argument *iwhence*.

PXFLSEEK allows the file offset to be set beyond the end of the existing end-of-file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFMKDIR

POSIX Subroutine: *Creates a new directory.*

Module

USE IFPOSIX

Syntax

CALL PXFMKDIR (*path*,*ilen*,*imode*,*ierror*)

path	(Input) Character. The path for the new directory.
ilen	(Input) INTEGER(4). The length of *path* string.
imode (L*X only)	(Input) INTEGER(4). The mode mask. Octal file-access mode.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFRMDIR
- PXFCHDIR

PXFMKFIFO (L*X, M*X)

POSIX Subroutine: *Creates a new FIFO.*

Module

USE IFPOSIX

Syntax

CALL PXFMKFIFO (*path*,*ilen*,*imode*,*ierror*)

path	(Input) Character. The path for the new FIFO.
ilen	(Input) INTEGER(4). The length of *path* string.
imode	(Input) INTEGER(4). The mode mask; specifies the FIFO's permissions. Octal file-access mode.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFMKFIFO subroutine creates a FIFO special file with name *path*. A FIFO special file is similar to a pipe, except that it is created in a different way. Once a FIFO special file is created, any process can open it for reading or writing in the same way as an ordinary file.

However, the FIFO file has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks it until some other process opens the same FIFO for writing, and vice versa.

See Also

- O to P
- PXFISFIFO

PXFOPEN

POSIX Subroutine: *Opens or creates a file.*

Module

USE IFPOSIX

Syntax

CALL PXFOPEN (*path*,*ilen*,*iopenflag*,*imode*,*ifildes*,*ierror*)

path	(Input) Character. The path of the file to be opened or created.
ilen	(Input) INTEGER(4). The length of *path* string.
iopenflag	(Input) INTEGER(4). The flags for the file. (For possible constant names that can be passed to PXFCONST or IPXFCONST, see below.)
imode	(Input) INTEGER(4). The permissions for a new file. This argument should always be specified when *iopenflag*=O_CREAT; otherwise, it is ignored. (For possible permissions, see below.)
ifildes	(Output) INTEGER(4). The returned file descriptor for the opened or created file.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

For *iopenflag*, you should specify one of the following constant values:

- O_RDONLY (read only)
- O_WRONLY (write only)
- O_RDWR (read and write)

In addition, you can also specify one of the following constant values by using a bitwise inclusive OR (IOR):

Value	Action
O_CREAT	Creates a file if the file does not exist.
O_EXCL	When used with O_CREAT, it causes the open to fail if the file already exists. In this case, a symbolic link exists, regardless of where it points to.
O_NOCTTY[1]	If *path* refers to a terminal device, it prevents it from becoming the process's controlling terminal even if the process does not have one.
O_TRUNC	If the file already exists, it is a regular file, and *imode* allows writing (its value is O_RDWR or O_WRONLY), it causes the file to be truncated to length 0.

Value	Action
O_APPEND	Opens the file in append mode. Before each write, the file pointer is positioned at the end of the file, as if with PXFLSEEK.
O_NONBLOCK (or O_NDELAY)[1]	When possible, opens the file in non-blocking mode. Neither the open nor any subsequent operations on the file descriptor that is returned will cause the calling process to wait. This mode need not have any effect on files other than FIFOs.
O_SYNC	Opens the file for synchronous I/O. Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware.
O_NOFOLLOW[1]	If *path* is a symbolic link, it causes the open to fail.
O_DIRECTORY[1]	If *path* is not a directory, it causes the open to fail.
O_LARGEFILE[1]	On 32-bit systems that support the Large Files System, it allows files whose sizes cannot be represented in 31 bits to be opened.
O_BINARY[2]	Opens the file in binary (untranslated) mode.
O_SHORT_LIVED[2]	Creates the file as temporary. If possible, it does not flush to the disk.
O_TEMPORARY[2]	Creates the file as temporary. The file is deleted when last file handle is closed.
O_RANDOM[2]	Specifies primarily random access from the disk.

3286

Value	Action
O_SEQUENTIAL[2]	Specifies primarily sequential access from the disk.
O_TEXT[2]	Opens the file in text (translated) mode.[3]

[1]L*X only

[2]W*32, W*64

[2]W*32, W*64

[3]For more information, see "Text and Binary Modes" in the Visual C++* programmer's guide.

Argument *imode* specifies the permissions to use if a new file is created. The permissions only apply to future accesses of the newly created file. The value for *imode* can be any of the following constant values (which can be obtained by using PXFCONST or IPXFCONST):

Value	Description
S_IRWXU	00700 user (file owner) has read, write and execute permission.
S_IRUSR, S_IREAD	00400 user has read permission.
S_IWUSR, S_IWRITE	00200 user has write permission.
S_IXUSR, S_IEXEC	00100 user has execute permission.
S_IRWXG[1]	00070 group has read, write and execute permission.
S_IRGRP[1]	00040 group has read permission.
S_IWGRP[1]	00020 group has write permission.
S_IXGRP[1]	00010 group has execute permission.
S_IRWXO[1]	00007 others have read, write and execute permission.
S_IROTH[1]	00004 others have read permission.

Value	Description
S_IWOTH[1]	00002 others have write permission.
S_IXOTH[1]	00001 others have execute permission.
[1]L*X only	

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

The following call opens a file for writing only and if the file does not exist, it is created:

```
call PXFOPEN( "OPEN.OUT", &
              8, &
              IOR( IPXFCONST(O_WRONLY), IPXFCONST(O_CREAT) ), &
              IOR( IPXFCONST(S_IREAD), IPXFCONST(S_IWRITE) ) )
```

See Also

- O to P
- PXFCLOSE
- IPXFCONST
- PXFCONST

PXFOPENDIR

POSIX Subroutine: *Opens a directory and associates a stream with it.*

Module

USE IFPOSIX

Syntax

```
CALL PXFOPENDIR (dirname,lendirname,opendirid,ierror)
```

dirname	(Input) Character. The directory name.
lendirname	(Input) INTEGER(4). The length of *dirname* string.

opendirid	(Output) INTEGER(4). The returned ID for the directory.
ierror	(Output) INTEGER(4). The error status. If successful, *ierror* is set to zero; otherwise, an error code.

This subroutine opens a directory pointed to by the *dirname* argument and returns the ID of the directory into *opendirid*. After the call, this ID can be used by functions PXFREADDIR, PXFREWINDDIR, PXFCLOSEDIR.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFCLOSEDIR
- PXFREADDIR
- PXFREADDIR

PXFPATHCONF

POSIX Subroutine: *Gets the value for a configuration option of an opened file.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPATHCONF (path,ilen,name,ival,ierror)
```

path	(Input) Character. The path to the opened file.
ilen	(Input) INTEGER(4). The length of *path*.
name	(Input) INTEGER(4). The configurable option.
ival	(Input) INTEGER(4). The value of the configurable option.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPATHCONF subroutine gets a value for the configuration option named for the opened file with path *path*.

The configuration option, defined in *name*, can be obtained by using PXFCONST or IPXFCONST with one of the following constant names:

Constant	Action
_PC_LINK_MAX	Returns the maximum number of links to the file. If *path* refers to a directory, then the value applies to the whole directory.
_PC_MAX_CANON[1]	Returns the maximum length of a formatted input line; the *path* must refer to a terminal.
_PC_MAX_INPUT[1]	Returns the maximum length of an input line; the *path* must refer to a terminal.
_PC_NAME_MAX	Returns the maximum length of a filename in *path* that the process is allowed to create.
_PC_PATH_MAX	Returns the maximum length of a relative pathname when *path* is the current working directory.
_PC_PIPE_BUF	Returns the size of the pipe buffer; the *path* must refer to a FIFO.
_PC_CHOWN_RESTRICTED[1]	Returns nonzero if PXFCHOWN may not be used on this file. If *path* refers to a directory, then this applies to all files in that directory.
_PC_NO_TRUNC[1]	Returns nonzero if accessing filenames longer than _POSIX_NAME_MAX will generate an error.
_PC_VDISABLE[1]	Returns nonzero if special character processing can be disabled; the *path* must refer to a terminal.

[1] L*X, M*X

On Linux* and Mac OS* X systems, the corresponding macros are defined in <unistd.h>. The values for *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in <unistd.h>. The following table shows the corresponding macro names for the above constants:

Constant	Corresponding Macro
_PC_LINK_MAX	_POSIX_LINK_MAX
_PC_MAX_CANON	_POSIX_MAX_CANON
_PC_MAX_INPUT	_POSIX_MAX_INPUT
_PC_NAME_MAX	_POSIX_NAME_MAX
_PC_PATH_MAX	_POSIX_PATH_MAX
_PC_PIPE_BUF	_POSIX_PIPE_BUF
_PC_CHOWN_RESTRICTED	_POSIX_CHOWN_RESTRICTED
_PC_NO_TRUNC	_POSIX_NO_TRUN C
_PC_VDISABLE	_POSIX_VDISABLE

See Also

- O to P
- IPXFCONST
- PXFCONST
- PXFFPATHCONF

PXFPAUSE

POSIX Subroutine: *Suspends process execution.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPAUSE (ierror)
```

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPAUSE subroutine causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFPIPE

POSIX Subroutine: *Creates a communications pipe between two processes.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPIPE (ireadfd,iwritefd,ierror)
```

ireadfd (Output) INTEGER(4). The file descriptor for reading.

iwritefd (Output) INTEGER(4). The file descriptor for writing.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFPIPE subroutine returns a pair of file descriptors, pointing to a pipe inode, and places them into *ireadfd* for reading and into *iwritefd* for writing.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFPOSIXIO

POSIX Subroutine: *Sets the current value of the POSIX I/O flag.*

Module

USE IFPOSIX

Syntax

```
CALL PXFPOSIXIO (new,old,ierror)
```

new	(Input) INTEGER(4). The new value for the POSIX I/O flag.
old	(Output) INTEGER(4). The previous value of the POSIX I/O flag.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

This subroutine sets the current value of the Fortran POSIX I/O flag and returns the previous value of the flag. The initial state of the POSIX I/O flag is unspecified.

If a file is opened with a Fortran OPEN statement when the value of the POSIX I/O flag is 1, the unit is accessed as if the records are newline delimited, even if the file does not contain records that are delimited by a new line character.

If a file is opened with a Fortran OPEN statement when the value of the POSIX I/O flag is zero, a connection to a file descriptor is not assumed and the records in the file are not required to be accessed as if they are newline delimited.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFPUTC

POSIX Subroutine: *Outputs a character to logical unit 6 (stdout).*

Module

USE IFPOSIX

Syntax

```
CALL PXFPUTC (ch,ierror)
```

ch	(Input) Character. The character to be written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code. A possible error code is EEND if the end of the file has been reached.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFGETC

PXFREAD

POSIX Subroutine: *Reads from a file.*

Module

USE IFPOSIX

Syntax

`CALL PXFREAD (ifildes,buf,nbyte,nread,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor of the file to be read from.
buf	(Output) Character. The buffer that stores the data read from the file.
nbyte	(Input) INTEGER(4). The number of bytes to read.
nread	(Output) INTEGER(4). The number of bytes that were read.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFREAD subroutine reads *nbyte* bytes from the file specified by *ifildes* into memory in *buf*. The subroutine returns the total number of bytes read into *nread*. If no error occurs, the value of *nread* will equal the value of *nbyte*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFWRITE

PXFREADDIR

POSIX Subroutine: *Reads the current directory entry.*

Module

USE IFPOSIX

Syntax

`CALL PXFREADDIR (idirid,jdirent,ierror)`

idirid	(Input) INTEGER(4). The ID of a directory obtained from PXFOPENDIR.
jdirent	(Output) INTEGER(4). A handle of structure `dirent`.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFREADDIR subroutine reads the entry of the directory associated with *idirid* into the structure associated with handle *jdirent*.

NOTE. To get a handle for an instance of the `dirent` structure, use PXFSTRUCTCREATE with the string 'dirent' for the structure name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFOPENDIR
- PXFREWINDDIR

PXFRENAME

POSIX Subroutine: *Changes the name of a file.*

Module

USE IFPOSIX

Syntax

CALL PXFRENAME (*old*,*lenold*,*new*,*lennew*,*ierror*)

old	(Input) Character. The name of the file to be renamed.
lenold	(Input) INTEGER(4). The length of *old* string.
new	(Input) Character. The new file name.
lennew	(Input) INTEGER(4). The length of *new* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFREWINDDIR

POSIX Subroutine: *Resets the position of the stream to the beginning of the directory.*

Module

USE IFPOSIX

Syntax

CALL PXFREWINDDIR (*idirid*,*ierror*)

idirid	(Input) INTEGER(4). The ID of a directory obtained from PXFOPENDIR.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFRMDIR

POSIX Subroutine: *Removes a directory.*

Module

USE IFPOSIX

Syntax

CALL PXFRMDIR (*path*,*ilen*,*ierror*)

path	(Input) Character. The directory to be removed. It must be empty.
ilen	(Input) INTEGER(4). The length of *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFMKDIR
- PXFCHDIR

PXFSETENV

POSIX Subroutine: *Adds a new environment variable or sets the value of an environment variable.*

Module

USE IFPOSIX

Syntax

CALL PXFSETENV (*name*,*lenname*,*new*,*lennew*,*ioverwrite*,*ierror*)

name	(Input) Character. The name of the environment variable.
lenname	(Input) INTEGER(4). The length of *name*.
new	(Input) Character. The value of the environment variable.

lennew	(Input) INTEGER(4). The length of *new*.
ioverwrite	(Input) INTEGER(4). A flag indicating whether to change the value of the environment variable if it exists.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If *name* does not exist, PXFSETENV adds it with value*new*.

If *name* exists, PXFSETENV sets its value to *new* if*ioverwrite* is a nonzero number. If *ioverwrite* is zero, the value of *name* is not changed.

If *lennew* is equal to zero, PXFSETENV sets the value of the environment variable to a string equal to *new* after removing any leading or trailing blanks.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program test2
use ifposix
character*10 name, new
integer lenname, lennew, ioverwrite, ierror
name = "FOR_NEW"
lenname = 7
new = "ON"
lennew = 2
ioverwrite = 1
CALL PXFSETENV (name, lenname, new, lennew, ioverwrite, ierror)
print *, "name= ", name
print *, "lenname= ", lenname
print *, "new= ", lenname
print *, "lennew= ", lenname
print *, "ierror= ", ierror
end
```

See Also

- O to P
- PXFGETENV

PXFSETGID (L*X, M*X)

POSIX Subroutine: *Sets the effective group ID of the current process.*

Module

USE IFPOSIX

Syntax

CALL PXFSETGID (*igid*,*ierror*)

igid	(Input) INTEGER(4). The group ID.
ierror	(Output) INTEGER(4). The error status. If successful, *ierror* is set to zero; otherwise, an error code. If the caller is the superuser, the real and saved group ID's are also set. This feature allows a program other than root to drop all of its group privileges, do some un-privileged work, and then re-engage the original effective group ID in a secure manner.

CAUTION. If the user is root then special care must be taken. PXFSETGID checks the effective gid of the caller. If it is the superuser, all process-related group ID's are set to gid. After this has occurred, it is impossible for the program to regain root privileges.

See Also

- O to P
- PXFGETGID

PXFSETPGID (L*X, M*X)

POSIX Subroutine: *Sets the process group ID.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSETPGID (ipid,ipgid,ierror)
```

ipid	(Input) INTEGER(4). The process group ID to change.
ipgid	(Input) INTEGER(4). The new process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSETPGID subroutine sets the process group ID of the process specified by *ipid* to *ipgid*.

If *ipid* is zero, the process ID of the current process is used. If *ipgid* is zero, the process ID of the process specified by *ipid* is used.

PXFSETPGID can be used to move a process from one process group to another, but both process groups must be part of the same session. In this case, *ipgid* specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.

PXFSETSID (L*X, M*X)

POSIX Subroutine: *Creates a session and sets the process group ID.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSETSID (isid,ierror)
```

isid	(Output) INTEGER(4). The session ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSETSID subroutine creates a new session if the calling process is not a process group leader.

The calling process is the leader of the new session and the process group leader for the new process group. The calling process has no controlling terminal.

The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session.

PXFSETUID (L*X, M*X)

POSIX Subroutine: *Sets the effective user ID of the current process.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSETUID (iuid,ierror)
```

iuid	(Output) INTEGER(4). The session ID.

ierror	(Output) INTEGER(4). The user status.

If successful, *ierror* is set to zero; otherwise, an error code.

If the effective user ID of the caller is root, the real and saved user ID's are also set. This feature allows a program other than root to drop all of its user privileges, do some un-privileged work, and then re-engage the original effective user ID in a secure manner.

CAUTION. If the user is root then special care must be taken. PXFSETUID checks the effective uid of the caller. If it is the superuser, all process-related user ID's are set to uid. After this has occurred, it is impossible for the program to regain root privileges.

See Also

- O to P
- PXFGETUID

PXFSIGACTION

POSIX Subroutine: *Changes the action associated with a specific signal. It can also be used to examine the action of a signal.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGACTION (isig,jsigact,josigact,ierror)
```

isig	(Input) INTEGER(4). The signal number whose action should be changed.
jsigact	(Input) INTEGER(4). A handle of structure `sigaction`. Specifies the new action for signal *isig*.
josigact	(Output) INTEGER(4). A handle of structure `sigaction`.Stores the previous action for signal *isig*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The signal specified in isig can be any valid signal except SIGKILL and SIGSTOP.

If *jsigact* is nonzero, the new action for signal *isig* is installed from the structure associated with handle *jsigact*. If *josigact* is nonzero, the previous action of the specified signal is saved in the structure associated with handle *josigact* where it can be examined.

On Windows* systems, PXFSIGACTION ignores the fields sa_mask and sa_flags in structure sigaction.

NOTE. To get a handle for an instance of the sigaction structure, use PXFSTRUCTCREATE with the string 'sigaction' for the structure name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFSTRUCTCREATE

PXFSIGADDSET (L*X, M*X)

POSIX Subroutine: *Adds a signal to the signal set.*

Module

USE IFPOSIX

Syntax

CALL PXFSIGADDSET (*jsigset*,*isigno*,*ierror*)

jsigset	(Input) INTEGER(4). A handle of structure sigset. This is the set to add the signal to.
isigno	(Input) INTEGER(4). The signal number to add to the set.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGADDSET subroutine adds signal number *isigno* to the set of signals associated with handle *jsigset*. This set of signals is used by PXFSIGACTION as field sa_mask in structure sigaction. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure sigaction).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE. To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFSIGDELSET
- PXFSIGACTION

PXFSIGDELSET (L*X, M*X)

POSIX Subroutine: *Deletes a signal from the signal set.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGDELSET (jsigset,isigno,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to delete the signal from.
isigno	(Input) INTEGER(4). The signal number to delete from the set.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGDELSET subroutine removes signal number *isigno* from the set of signals associated with handle *jsigset*. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE. To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFSIGADDSET
- PXFSIGACTION

PXFSIGEMPTYSET (L*X, M*X)

POSIX Subroutine: *Empties a signal set.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGEMPTYSET (jsigset,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to empty.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, nonzero.

The PXFSIGEMPTYSET subroutine initializes the signal set associated with handle *jsigset* to empty; all signals are excluded from the set. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE. To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFSIGFILLSET
- PXFSIGACTION

PXFSIGFILLSET (L*X, M*X)

POSIX Subroutine: *Fills a signal set.*

Module

USE IFPOSIX

Syntax

`CALL PXFSIGFILLSET (jsigset,ierror)`

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set to fill.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGFILLSET subroutine initializes the signal set associated with handle *jsigset* to full; all signals are included into the set. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE. To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFSIGEMPTYSET
- PXFSIGACTION

PXFSIGISMEMBER (L*X, M*X)

POSIX Subroutine: *Tests whether a signal is a member of a signal set.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGISMEMBER (jsignset,isigno,ismember,ierror)
```

jsignset	(Input) INTEGER(4). A handle of structure `sigset`. This is the set the signal will be tested in.
isigno	(Input) INTEGER(4). The signal number to test for membership.
ismember	(Output) Logical. The returned result.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGISMEMBER subroutine tests whether *isigno* is a member of the set associated with handle *jsignset*. If the signal is a member of the set, *ismember* is set to .TRUE.; otherwise, .FALSE.. This set of signals is used by PXFSIGACTION as field sa_mask in structure `sigaction`. It defines the set of signals that will be blocked during execution of the signal handler function (the field sa_handler in structure `sigaction`).

On Windows* systems, PXFSIGACTION ignores the field sa_mask in structure `sigaction`.

NOTE. To get a handle for an instance of the `sigset` structure, use PXFSTRUCTCREATE with the string 'sigset' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFSIGACTION

PXFSIGPENDING (L*X, M*X)

POSIX Subroutine: *Examines pending signals.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGPENDING (jsigset,ierror)
```

jsigset	(Input) INTEGER(4). A handle of structure sigaction. The signals to examine.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSIGPENDING subroutine is used to examine pending signals (ones that have been raised while blocked). The signal mask of the pending signals is stored in the signal set associated with handle *jsigset*.

PXFSIGPROCMASK (L*X, M*X)

POSIX Subroutine: *Changes the list of currently blocked signals.*

Module

USE IFPOSIX

Syntax

CALL PXFSIGPROCMASK (*ihow*,*jsigset*,*josigset*,*ierror*)

ihow	(Input) INTEGER(4). Defines the action for *jsigset*.
jsigset	(Input) INTEGER(4). A handle of structure sigset. The signals to examine.
josigset	(Input) INTEGER(4). A handle of structure sigset. Stores the previous mask of blocked signals.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The argument *ihow* indicates the way in which the set is to be changed, and consists of one of the following constant names:

Constant [1]	Action
SIG_BLOCK	The resulting set of blocked signals will be the union of the current signal set and the *jsigset* signal set.

Constant [1]	Action
SIG_UNBLOCK	The resulting set of blocked signals will be the current set of blocked signals with the signals in *jsigset* removed. It is legal to attempt to unblock a signal that is not blocked.
SIG_SETMASK	The resulting set of blocked signals will be the *jsigset* signal set.

[1]These names can be used in PXFCONST or IPXFCONST.

If *josigset* is non-zero, the previous value of the signal mask is stored in the structure associated with handle *josigset*.

See Also

- O to P
- IPXFCONST
- PXFCONST

PXFSIGSUSPEND (L*X, M*X)

POSIX Subroutine: *Suspends the process until a signal is received.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSIGSUSPEND (jsigset,ierror)
```

jsigset	(Input) INTEGER(4). A handle of structure `sigset`. Specifies a set of signals.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFSIGSUSPEND temporarily replaces the signal mask for the process with that given by the structure associated with the *jsigset* handle; it then suspends the process until a signal is received.

PXFSLEEP

POSIX Subroutine: *Forces the process to sleep.*

Module

USE IFPOSIX

Syntax

CALL PXFSLEEP (*iseconds*,*isecleft*,*ierror*)

iseconds	(Input) INTEGER(4). The number of seconds to sleep.
isecleft	(Output) INTEGER(4). The number of seconds left to sleep.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSLEEP subroutine forces the current process to sleep until seconds *iseconds* have elapsed or a signal arrives that cannot be ignored.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFSTAT

POSIX Subroutine: *Gets a file's status information.*

Module

USE IFPOSIX

Syntax

CALL PXFSTAT (*path*,*ilen*,*jstat*,*ierror*)

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of *path* string.
jstat	(Input) INTEGER(4). A handle of structure stat.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSTAT subroutine puts the status information for the file specified by path into the structure associated with handle *jstat*.

NOTE. To get a handle for an instance of the `stat` structure, use PXFSTRUCTCREATE with the string 'stat' for the structure name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P
- PXFSTRUCTCREATE

PXFSTRUCTCOPY

POSIX Subroutine: *Copies the contents of one structure to another.*

Module

USE IFPOSIX

Syntax

```
CALL PXFSTRUCTCOPY (structname,jhandle1,jhandle2,ierror)
```

structname	(Input) Character. The name of the structure.
jhandle1	(Input) INTEGER(4). A handle to the structure to be copied.
jhandle2	(Input) INTEGER(4). A handle to the structure that will receive the copy.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFSTRUCTCREATE

PXFSTRUCTCREATE

POSIX Subroutine: *Creates an instance of the specified structure.*

Module

USE IFPOSIX

Syntax

CALL PXFSTRUCTCREATE (*structname*,*jhandle*,*ierror*)

structname	(Input) Character. The name of the structure. As for any character string, the name must be specified in single or double quotes; for example, the structure `sigaction` would be specified as 'sigaction'. (For more information on available structures, see below.)
jhandle	(Output) INTEGER(4). The handle of the newly-created structure.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

If your application passes information to the system, you should call one of the PXF(type)SET subroutines. If your application needs to get information from the structure, you should call one of the PXF(type)GET subroutines.

The following table shows:

- The structures that are available in the Fortran POSIX library
- The fields within each structure
- The subroutines you must use to access the structure fields

Structure Name	Field Names	Subroutines for Access
sigset [1]	Fields are hidden.	PXFSIGEMPTYSET [1], PXFSIGFILLSET [1], PXFSIGADDSET [1], or PXFSIGDELSET [1]

Structure Name	Field Names	Subroutines for Access
sigaction	sa_handler	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	sa_mask	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	sa_flags	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
utsname	sysname	For all fields:
	nodename	PXFSTRGET
	release	
	version	
	machine	
tms	tms_utime	For all fields:
	tms_stime	PXFINTGET or PXFINT8GET
	tms_cutime	
	tms_cstime	
dirent	d_name	PXFSTRGET
stat	st_mode	For all fields:
	st_ino	PXFINTGET or PXFINT8GET
	st_dev	
	st_nlink	
	st_uid	
	st_gid	
	st_size	
	st_atime	
	st_mtime	
	st_ctime	

Structure Name	Field Names	Subroutines for Access
utimbuf	actime	For all fields:
	modtime	PXFINTGET or PXFINT8GET
flock [1]	l_type	For all fields:
	l_whence	PXFINTGET or PXFINT8GET
	l_start	
	l_len	
	l_pid	
termios [1]	c_iflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_oflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_cflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_lflag	PXFINTGET/PXFINTSET or PXFINT8GET/PXFINT8SET
	c_cc	PXFAINTGET/PXFAINTSET or PXFAINT8GET/PXFAINT8SET
group [1]	gr_name	PXFSTRGET
	gr_gid	PXFINTGET or PXFINT8GET
	gr_nmem	PXFINTGET or PXFINT8GET
	gr_mem	PXFESTRGET
passwd [1]	pw_name	PXFSTRGET
	pw_uid	PXFINTGET or PXFINT8GET
	pw_gid	PXFINTGET or PXFINT8GET
	pw_dir	PXFSTRGET
	pw_shell	PXFSTRGET

3314

Structure Name	Field Names	Subroutines for Access

[1]L*X only

As for any character string, you must use single or double quotes when specifying a field name in a PXF(type)GET or PXF(type)SET subroutine. For example, field name sysname (in structure `utsname`) must be specified as 'sysname'.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program test4
  use ifposix
  implicit none
  integer(jhandle_size) jhandle1,jhandle2
  integer(4) ierror,ilen1
  print *," Create a first instance for structure 'utsname' "
  call PXFSTRUCTCREATE("utsname",jhandle1,ierror)
  if(ierror.NE.0) STOP 'Error: cannot create structure for jhandle1'
  print *," Create a second instance for structure 'utsname' "
  call PXFSTRUCTCREATE("utsname",jhandle2,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jhandle1,ierror)
     STOP 'test failed - cannot create structure for jhandle2'
  end if
  print *,"Fill the structure associated with jhandle1 with arbitrary data"
  call PXFSTRSET(jhandle1,"sysname","00000000000000",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component sysname for jhandle1')
  call PXFSTRSET(jhandle1,"Nodename","11111111111111",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component nodename for jhandle1')
  call PXFSTRSET(jhandle1,"RELEASE","22222222222222",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component release for jhandle1')
  call PXFSTRSET(jhandle1,"verSION","33333333333333",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component version for jhandle1')
  call PXFSTRSET(jhandle1,"machine","44444444444444",14,ierror)
  if(ierror.NE.0) call Error('Error: can't set component machine for jhandle1')
  print *,"Fill the structure associated with jhandle2 with arbitary data"
  call PXFSTRSET(jhandle2,"sysname","aaaaaaaaa",7,ierror)
```

```
if(ierror.NE.0) call Error('Error: can't set component sysname for jhandle2')
call PXFSTRSET(jhandle2,"Nodename","BBBBBBBBB  BBB",14,ierror)
if(ierror.NE.0) call Error('Error: can't set component nodename for jhandle2')
call PXFSTRSET(jhandle2,"RELEASE","cCCC cc-cccnc",12,ierror)
if(ierror.NE.0) call Error('Error: can't set component release for jhandle2')
call PXFSTRSET(jhandle2,"verSION","ddddd",1,ierror)
if(ierror.NE.0) call Error('Error: can't set component version for jhandle2')
call PXFSTRSET(jhandle2,"machine","eeeeeee",6,ierror)
if(ierror.NE.0) call Error('Error: can't set component machine for jhandle2')
print *,"Print contents of the structure associated with jhandle1"
call PRINT_UTSNAME(jhandle1)
print *,"Print contents of the structure associated with jhandle2"
call PRINT_UTSNAME(jhandle2)
print *,"Get operating system info into structure associated with jhandle1"
call PXFUNAME(jhandle1,ierror)
if(ierror.NE.0) call Error('Error: call to PXFUNAME has failed')
print *,"Print contents of the structure associated with jhandle1"
print*,"  returned from PXFUNAME"
call PRINT_UTSNAME(jhandle1)
print *,"Copy the contents of the structure associated with jhandle1"
print *,"  into the structure associated with jhandle2"
call PXFSTRUCTCOPY("utsname",jhandle1,jhandle2,ierror)
if(ierror.NE.0) call Error('Error: can't copy jhandle1 contents into jhandle2')
print *,"Print the contents of the structure associated with jhandle2."
print *,"  It should be the same after copying."
call PRINT_UTSNAME(jhandle2)
print *,"Free memory for instance of structure associated with jhandle1"
call PXFSTRUCTFREE(jhandle1,ierror)
```

```
  if(ierror.NE.0) STOP 'Error: can't free instance of structure for jhandle1'
  print *,"Free memory for instance of structure associated with jhandle2"
  call PXFSTRUCTFREE(jhandle2,ierror)
  if(ierror.NE.0) STOP 'Error: can't free instance of structure for jhandle2'
  print *,"Program terminated normally"
  call PXFEXIT(0)
end
```

See Also

- O to P
- PXFSTRUCTFREE
- the example in PXFTIMES

PXFSTRUCTFREE

POSIX Subroutine: *Deletes the instance of a structure.*

Module

USE IFPOSIX

Syntax

CALL PXFSTRUCTFREE (*jhandle*,*ierror*)

jhandle	(Input) INTEGER(4). The handle of a structure.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFSTRUCTFREE subroutine deletes the instance of the structure associated with handle *jhandle*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFSTRUCTCREATE, the example in PXFTIMES

PXFSYSCONF

POSIX Subroutine: *Gets values for system limits or options.*

Module

USE IFPOSIX

Syntax

CALL PXFSYSCONF (*name*,*ival*,*ierror*)

name	(Input) INTEGER(4). The system option you want information about.
ival	(Output) INTEGER(4). The returned value.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFSYSCONF lets you determine values for system limits or system options at runtime.

The value for *name* can be any of the following constants:

Constant	Description
_SC_ARG_MAX[1]	Indicates the maximum length of the arguments to the PXFEXEC family of routines.
_SC_CHILD_MAX[1]	Indicates the number of simultaneous processes per user ID.
_SC_CLK_TCK	Indicates the number of clock ticks per second.
_SC_STREAM_MAX[2]	Indicates the maximum number of streams that a process can have open at any time.
_SC_TZNAME_MAX	Indicates the maximum number of bytes in a timezone name.
_SC_OPEN_MAX	Indicates the maximum number of files that a process can have open at any time.

Constant	Description
_SC_JOB_CONTROL[1]	Indicates whether POSIX-style job control is supported.
_SC_SAVED_IDS[1]	Indicates whether a process has a saved set-user-ID and a saved set-group-ID.
_SC_VERSION[1]	Indicates the year and month the POSIX.1 standard was approved in the format YYYYMML; the value 199009L indicates the most recent revision, 1990.
_SC_BC_BASE_MAX[1]	Indicates the maximum obase value accepted by the bc(1) utility.
_SC_BC_DIM_MAX[1]	Indicates the maximum value of elements that bc(1) permits in an array.
_SC_BC_SCALE_MAX[1]	Indicates the maximum scale value allowed by bc(1).
_SC_BC_STRING_MAX[1]	Indicates the maximum length of a string accepted by bc(1).
_SC_COLL_WEIGHTS_MAX[1]	Indicates the maximum numbers of weights that can be assigned to an entry of the LC_COLLATE order keyword in the locale definition file.
_SC_EXPR_NEST_MAX[1,3]	Indicates the maximum number of expressions that can be nested within parentheses by expr(1).
_SC_LINE_MAX[1]	Indicates the maximum length of a utility's input line length, either from standard input or from a file. This includes the length for a trailing newline.

Constant	Description
_SC_RE_DUP_MAX[1]	Indicates the maximum number of repeated occurrences of a regular expression when the interval notation \{m,n\} is used.
_SC_2_VERSION[1]	Indicates the version of the POSIX.2 standard; it is in the format YYYYMML.
_SC_2_DEV[1]	Indicates whether the POSIX.2 C language development facilities are supported.
_SC_2_FORT_DEV[1]	Indicates whether the POSIX.2 FORTRAN language development utilities are supported.
_SC_2_FORT_RUN[1]	Indicates whether the POSIX.2 FORTRAN runtime utilities are supported.
_SC_2_LOCALEDEF[1]	Indicates whether the POSIX.2 creation of locates via localedef(1) is supported.
_SC_2_SW_DEV[1]	Indicates whether the POSIX.2 software development utilities option is supported.
_SC_PAGESIZE (or _SC_PAGE_SIZE)	Indicates the size of a page (in bytes).
_SC_PHYS_PAGES[4]	Indicates the number of pages of physical memory. Note that it is possible for the product of this value and the value of _SC_PAGE_SIZE to overflow.
_SC_AVPHYS_PAGES[4]	Indicates the number of currently available pages of physical memory.

[1]L*X only

[2]The corresponding POSIX macro is STREAM_MAX.

[3]The corresponding POSIX macro is EXPR_NEST_MAX.

[4]L*X, W*32, W*64

The corresponding macros are defined in <bits/confname.h> on Linux* systems; <unistd.h> on Mac OS* X systems. The values for argument *name* can be obtained by using PXFCONST or IPXFCONST when passing the string names of predefined macros in the appropriate .h file.

See Also

- O to P
- IPXFCONST
- PXFCONST

PXFTCDRAIN (L*X, M*X)

POSIX Subroutine: *Waits until all output written has been transmitted.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCDRAIN (ifildes,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFTCFLOW (L*X, M*X)

POSIX Subroutine: *Suspends the transmission or reception of data.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCFLOW (ifildes,iaction,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
iaction	(Input) INTEGER(4). The action to perform.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCFLOW subroutine suspends or resumes transmission or reception of data from the terminal referred to by *ifildes*. The action performed depends on the value of *iaction*, which must be one of the following constant names:

Constant [1]	Action
TCOOFF	Output is suspended.
TCOON	Output is resumed.
TCIOFF	A STOP character is transmitted. This should cause the terminal to stop transmitting data to the system.
TCION	A START character is transmitted. This should cause the terminal to resume transmitting data to the system.

[1]These names can be used in PXFCONST or IPXFCONST.

See Also

- O to P
- IPXFCONST
- PXFCONST

PXFTCFLUSH (L*X, M*X)

POSIX Subroutine: *Discards terminal input data, output data, or both.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCFLUSH (ifildes,iaction,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
iaction	(Input) INTEGER(4). The action to perform.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The action performed depends on the value of *iaction*, which must be one of the following constant names:

Constant [1]	Action
TCIFLUSH	Discards all data that has been received but not read.
TCOFLUSH	Discards all data that has been written but not transmitted.
TCIOFLUSH	Discards both data received but not read and data written but not transmitted. (Performs TCIFLUSH and TCOFLUSH actions.)

[1]These names can be used in PXFCONST or IPXFCONST.

See Also

- O to P
- IPXFCONST
- PXFCONST

PXFTCGETATTR (L*X, M*X)

POSIX Subroutine: *Reads current terminal settings.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCGETATTR (ifildes,jtermios,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
jtermios	(Output) INTEGER(4). A handle for structure `termios`. Stores the terminal settings.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

NOTE. To get a handle for an instance of the `termios`structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFTCSETATTR

PXFTCGETPGRP (L*X, M*X)

POSIX Subroutine: *Gets the foreground process group ID associated with the terminal.*

Module

USE IFPOSIX

Syntax

`CALL PXFTCGETPGRP (`*ifildes*`,`*ipgid*`,`*ierror*`)`

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ipgid	(Output) INTEGER(4). The returned process group ID.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

- O to P
- PXFTCSETPGRP

PXFTCSENDBREAK (L*X, M*X)

POSIX Subroutine: *Sends a break to the terminal.*

Module

USE IFPOSIX

Syntax

`CALL PXFTCSENDBREAK (ifildes,iduration,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
iduration	(Input) INTEGER(4). Indicates how long the break should be.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCSENDBREAK subroutine sends a break (a '\0' with a framing error) to the terminal associated with *ifildes*.

PXFTCSETATTR (L*X, M*X)

POSIX Subroutine: *Writes new terminal settings.*

Module

USE IFPOSIX

Syntax

`CALL PXFTCSETATTR (ifildes,ioptacts,jtermios,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ioptacts	(Input) INTEGER(4). Specifies when the terminal changes take effect.
jtermios	(Input) INTEGER(4). A handle for structure `termios`. Contains the new terminal settings.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTCSETATTR subroutine copies all terminal parameters from structure `termios`into the terminal associated with *`ifildes`*. When the terminal settings will change depends on the value of *`ioptacts`*, which must be one of the following constant names:

Constant [1]	Action
TCSANOW	The changes occur immediately.
TCSADRAIN	The changes occur after all output written to *`ifildes`* has been transmitted.
TCSAFLUSH	The changes occur after all output written to *`ifildes`* has been transmitted, and all input that had been received but not read has been discarded.

[1]These names can be used in PXFCONST or IPXFCONST.

NOTE. To get a handle for an instance of the `termios`structure, use PXFSTRUCTCREATE with the string 'termios' for the structure name.

See Also

- O to P
- PXFSTRUCTCREATE
- PXFTCGETATTR

PXFTCSETPGRP (L*X, M*X)

POSIX Subroutine: *Sets the foreground process group ID associated with the terminal.*

Module

USE IFPOSIX

Syntax

```
CALL PXFTCSETPGRP (ifildes,ipgid,ierror)
```

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
ipgid	(Input) INTEGER(4). The foreground process group ID for *ifildes*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

See Also

- O to P
- PXFTCGETPGRP

PXFTIME

POSIX Subroutine: *Returns the current system time.*

Module

USE IFPOSIX

Syntax

CALL PXFTIME (*itime*,*ierror*)

itime	(Output) INTEGER(4). The returned system time.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTIME subroutine returns the number of seconds since Epoch (00:00:00 UTC, January 1, 1970).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFTIMES.

PXFTIMES

POSIX Subroutine: *Returns process times.*

Module

USE IFPOSIX

Syntax

`CALL PXFTIMES (jtms,itime,ierror)`

jtms	(Output) INTEGER(4). A handle of structure `tms`.
itime	(Output) INTEGER(4). The returned time since system startup.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFTIMES subroutine fills the fields of structure `tms` associated with handle *jtms* with components of time that was spent by the current process. The structure fields are:

- tms_utime - User CPU time
- tms_stime - System CPU time
- tms_cutime - User time of child process
- tms_cstime - System time of child process

All members are measured in system clocks. The values can be converted to seconds by dividing by value *ival* returned from the following call:

```
PXFSYSCONF(IPXFCONST('_SC_CLK_TCK'), ival, ierror)
```

User time is the time charged for the execution of user instructions of the calling process. System time is the time charged for execution by the system on behalf of the calling process.

NOTE. To get a handle for an instance of the `tms` structure, use PXFSTRUCTCREATE with the string 'tms' for the structure name.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
program test_uname
  use ifposix
  implicit none
  integer(jhandle_size) jtms1, jtms2
  integer(4) ierror,i
  integer(4),parameter :: n=10000000
  integer(SIZEOF_CLOCK_T) itime,time1,time2, user_time1,user_time2
  integer(SIZEOF_CLOCK_T) system_time1,system_time2
  integer(4) clocks_per_sec, iname
  real(8) s, PI
  real(8) seconds_user, seconds_system
  print *,"Create a first instance for structure 'tms'"
  call PXFSTRUCTCREATE("tms",jtms1,ierror)
  if(ierror.NE.0) STOP 'Error: cannot create structure for handle jtms1'
  print *,"Create a second instance for structure 'tms'"
  call PXFSTRUCTCREATE("tms",jtms2,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     STOP 'Error: cannot create structure for handle jtms2'
  end if
  print *, 'Do some calculations'
  call PXFTIMES(jtms1, itime,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the first call of PXFTIMES fails'
  end if
```

```
  call PXFTIME(time1, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the first call of PXFTIME fails'
  end if
  s = 0._8
  PI = atan(1._8)*4
  do i=0, n
    s = s + cos(i*PI/n)*sin(i*PI/n)
  end do
  print *," s=",s
  call PXFTIMES(jtms2, itime,ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the second call of PXFTIMES fails'
  end if
  call PXFTIME(time2, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the second call of PXFTIME fails'
  end if
!DEC$ IF DEFINED(_M_IA64)
     call PXFINT8GET(jtms1,"tms_utime",user_time1,ierror)
     call PXFINT8GET(jtms1,"tms_stime",system_time1,ierror)
     call PXFINT8GET(jtms2,"tms_utime",user_time2,ierror)
```

```
      call PXFINT8GET(jtms2,"tms_stime",system_time2,ierror)
!DEC$ ELSE
      call PXFINTGET(jtms1,"tms_utime",user_time1,ierror)
      call PXFINTGET(jtms1,"tms_stime",system_time1,ierror)
      call PXFINTGET(jtms2,"tms_utime",user_time2,ierror)
      call PXFINTGET(jtms2,"tms_stime",system_time2,ierror)
!DEC$ ENDIF
  iname = IPXFCONST("_SC_CLK_TCK")
  call PXFSYSCONF(iname,clocks_per_sec, ierror)
  if(ierror.NE.0) then
     call PXFSTRUCTFREE(jtms1,ierror)
     call PXFSTRUCTFREE(jtms2,ierror)
     STOP 'Error: the call of PXFSYSCONF fails'
  end if
  seconds_user = (user_time2 - user_time1)/DBLE(clocks_per_sec)
  seconds_system = (system_time2 - system_time1)/DBLE(clocks_per_sec)
  print *," The processor time of calculations:"
  print *," User code execution(in seconds):", seconds_user
  print *," Kernal code execution(in seconds):", seconds_system
  print *," Total processor time(in seconds):", seconds_user + seconds_system
  print *," Elapsed wall clock time(in seconds):", time2 - time1
  print *,"Free memory for instance of structure associated with jtms"
  call PXFSTRUCTFREE(jtms1,ierror)
  call PXFSTRUCTFREE(jtms2,ierror)
end program
```

See Also

- O to P
- PXFSTRUCTCREATE

PXFTTYNAM (L*X, M*X)

POSIX Subroutine: *Gets the terminal pathname.*

Module

USE IFPOSIX

Syntax

CALL PXFTTYNAM (*ifildes*,*s*,*ilen*,*ierror*)

ifildes	(Input) INTEGER(4). The file descriptor associated with the terminal.
s	(Output) Character. The returned terminal pathname.
ilen	(Output) INTEGER(4). The length of the string stored in *s*.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

PXFUCOMPARE

POSIX Subroutine: *Compares two unsigned integers.*

Module

USE IFPOSIX

Syntax

CALL PXFUCOMPARE (*i1*,*i2*,*icmpr*,*idiff*)

i1, *i2* (Input) INTEGER(4). The two unsigned integers to compare.

icmpr (Output) INTEGER(4). The result of the comparison; one of the following values:

-1	If *i1* < *i2*
0	If *i1* = *i2*
1	If *i1* > *i2*

idiff (Output) INTEGER(4). The absolute value of the difference.

The PXFUCOMPARE subroutine compares two unsigned integers and returns the absolute value of their difference into *idiff*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFUMASK

POSIX Subroutine: *Sets a new file creation mask and gets the previous one.*

Module

USE IFPOSIX

Syntax

CALL PXFUMASK (*icmask*,*iprevcmask*,*ierror*)

icmask	(Input) INTEGER(4). The new file creation mask.
iprevcmask	(Output) INTEGER(4). The previous file creation mask.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFUNAME

POSIX Subroutine: *Gets the operation system name.*

Module

USE IFPOSIX

Syntax

CALL PXFUNAME (*jutsname*,*ierror*)

jutsname	(Input) INTEGER(4). A handle of structure utsname.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFUNAME subroutine provides information about the operation system. The information is stored in the structure associated with handle *jutsname*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in PXFSTRUCTCREATE

PXFUNLINK

POSIX Subroutine: *Removes a directory entry.*

Module

USE IFPOSIX

Syntax

CALL PXFUNLINK (*path*,*ilen*,*ierror*)

path	(Input) Character. The name of the directory entry to remove.
ilen	(Input) INTEGER(4). The length of *path* string.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFUTIME

POSIX Subroutine: *Sets file access and modification times.*

Module

USE IFPOSIX

Syntax

CALL PXFUTIME (*path*,*ilen*,*jutimbuf*,*ierror*)

path	(Input) Character. The path to the file.
ilen	(Input) INTEGER(4). The length of *path* string.
jutimbuf	(Input) INTEGER(4). A handle of structure utimbuf.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFUTIME subroutine sets access and modification times for the file pointed to by path. The time values are retrieved from structure *utimbuf*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

PXFWAIT (L*X, M*X)

POSIX Subroutine: *Waits for a child process.*

Module

USE IFPOSIX

Syntax

CALL PXFWAIT (*istat*,*iretpid*,*ierror*)

istat	(Output) INTEGER(4). The returned status of the child process.
iretpid	(Output) INTEGER(4). The process ID of the stopped child process.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWAIT subroutine suspends execution of the current process until a child has exited, or until a signal is delivered whose action terminates the current process or calls a signal handling routine. If the child has already exited by the time of the call (a "zombie" process), a return is immediately made. Any system resources used by the child are freed.

The subroutine returns in *iretpid* the value of the process ID of the child that exited, or zero if no child was available. The returned value in *istat* can be used in subroutines IPXFWEXITSTATUS, IPXFWSTOPSIG, IPXFWTERMSIG, PXFWIFEXITED, PXFWIFSIGNALLED, and PXFWIFSTOPPED.

Example

```
program t1
use ifposix
integer(4) ipid, istat, ierror, ipid_ret, istat_ret
  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if
  end if
10 FORMAT (A,Z)
```

```
end program
```

See Also

- O to P
- PXFWAITPID
- IPXFWEXITSTATUS
- IPXFWSTOPSIG
- IPXFWTERMSIG
- PXFWIFEXITED
- PXFWIFSIGNALLED
- PXFWIFSTOPPED

PXFWAITPID (L*X, M*X)

POSIX Subroutine: *Waits for a specific PID.*

Module

USE IFPOSIX

Syntax

CALL PXFWAITPID (*ipid*,*istat*,*ioptions*,*iretpid*,*ierror*)

ipid (Input) INTEGER(4). The PID to wait for. One of the following values:

Value	Action
< -1	Specifies to wait for any child process whose process group ID is equal to the absolute value of *ipid*.
-1	Specifies to wait for any child process; this is the same behavior as PXFWAIT.
0	Specifies to wait for any child process whose process group ID is equal to that of the calling process.

Value	Action
> 0	Specifies to wait for the child whose process ID is equal to the value of *ipid*.

istat (Output) INTEGER(4). The returned status of the child process.

ioptions (Input) INTEGER(4). One or more of the following constant values (which can be passed to PXFCONST or IPXFCONST):

Value	Action
WNOHANG	Specifies to return immediately if no child process has exited.
WUNTRACED	Specifies to return for child processes that have stopped, and whose status has not been reported.

iretpid (Output) INTEGER(4). The PID of the stopped child process.

ierror (Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWAITPID subroutine suspends execution of the current process until the child specified by *ipid* has exited, or until a signal is delivered whose action terminates the current process or calls a signal handling routine. If the child specified by *ipid* has already exited by the time of the call (a "zombie" process), a return is immediately made. Any system resources used by the child are freed.

The returned value in *istat* can be used in subroutines IPXFWEXITSTATUS, IPXFWSTOPSIG, IPXFWTERMSIG, PXFWIFEXITED, PXFWIFSIGNALLED, and PXFWIFSTOPPED.

See Also

- O to P
- PXFWAIT
- IPXFWEXITSTATUS
- IPXFWSTOPSIG
- IPXFWTERMSIG

- PXFWIFEXITED
- PXFWIFSIGNALLED
- PXFWIFSTOPPED

PXFWIFEXITED (L*X, M*X)

POSIX Function: *Determines if a child process has exited.*

Module

USE IFPOSIX

Syntax

`result = PXFWIFEXITED (istat)`

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has exited normally; otherwise, .FALSE..

Example

```
program t1

use ifposix

integer(4) ipid, istat, ierror, ipid_ret, istat_ret

  print *," the child process will be born"
  call PXFFORK(IPID, IERROR)
  call PXFGETPID(IPID_RET,IERROR)
```

```
  if(IPID.EQ.0) then
     print *," I am a child process"
     print *," My child's pid is", IPID_RET
     call PXFGETPPID(IPID_RET,IERROR)
     print *," The pid of my parent is",IPID_RET
     print *," Now I have exited with code 0xABCD"
     call PXFEXIT(Z'ABCD')
  else
     print *," I am a parent process"
     print *," My parent pid is ", IPID_RET
     print *," I am creating the process with pid", IPID
     print *," Now I am waiting for the end of the child process"
     call PXFWAIT(ISTAT, IPID_RET, IERROR)
     print *," The child with pid ", IPID_RET," has exited"
     if( PXFWIFEXITED(ISTAT) ) then
       print *, " The child exited normally"
       istat_ret = IPXFWEXITSTATUS(ISTAT)
       print 10," The low byte of the child exit code is", istat_ret
     end if

  end if
10 FORMAT (A,Z)

end program
```

See Also

- O to P
- PXFWIFSIGNALED
- PXFWIFSTOPPED

PXFWIFSIGNALED (L*X, M*X)

POSIX Function: *Determines if a child process has exited because of a signal.*

Module

USE IFPOSIX

Syntax

```
result = PXFWIFSIGNALED (istat)
```

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has exited because of a signal that was not caught; otherwise, .FALSE..

See Also

- O to P
- PXFWIFEXITED
- PXFWIFSTOPPED

PXFWIFSTOPPED (L*X, M*X)

POSIX Function: *Determines if a child process has stopped.*

Module

USE IFPOSIX

Syntax

```
result = PXFWIFSTOPPED (istat)
```

istat	(Output) INTEGER(4). The status of the child process (obtained from PXFWAIT or PXFWAITPID).

Results

The result type is logical. The result value is .TRUE. if the child process has stopped; otherwise, .FALSE..

See Also

- O to P
- PXFWIFEXITED
- PXFWIFSIGNALED

PXFWRITE

POSIX Subroutine: *Writes to a file.*

Module

USE IFPOSIX

Syntax

`CALL PXFWRITE (ifildes,buf,nbyte,nwritten,ierror)`

ifildes	(Input) INTEGER(4). The file descriptor for the file to be written to.
buf	(Input) Character. The buffer that contains the data to write into the file.
nbyte	(Input) INTEGER(4). The number of bytes to write.
nwritten	(Output) INTEGER(4). The returned number of bytes written.
ierror	(Output) INTEGER(4). The error status.

If successful, *ierror* is set to zero; otherwise, an error code.

The PXFWRITE subroutine writes *nbyte* bytes from the storage *buf* into a file specified by file descriptor *ifildes*. The subroutine returns the total number of bytes read into *nwritten*. If no error occurs, the value of *nwritten* will equal the value of *nbyte*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- O to P

- PXFREAD

Q to R

QCMPLX

Elemental Intrinsic Function (Specific):
Converts an argument to COMPLEX(16) type. This function cannot be passed as an actual argument.

Syntax

```
result = QCMPLX (x[,y])
```

x	(Input) Must be of type integer, real, or complex.
y	(Input; optional) Must be of type integer or real. It must not be present if *x* is of type complex.

Results

The result type is COMPLEX(16) (or COMPLEX*32).

If only one noncomplex argument appears, it is converted into the real part of the result value and zero is assigned to the imaginary part. If *y* is not specified and *x* is complex, the result value is CMPLX(REAL(*x*), AIMAG(*x*)).

If two noncomplex arguments appear, the complex value is produced by converting the first argument into the real part of the value, and converting the second argument into the imaginary part.

QCMPLX(*x*, *y*) has the complex value whose real part is REAL(*x*, *kind*=16) and whose imaginary part is REAL(*y*, *kind*=16).

Example

QCMPLX (-3) has the value (-3.0Q0, 0.0Q0).

QCMPLX (4.1, 2.3) has the value (4.1Q0, 2.3Q0).

See Also

- Q to R
- CMPLX
- DCMPLX

- FLOAT
- INT
- IFIX
- REAL
- SNGL

QEXT

Elemental Intrinsic Function (Generic):
Converts a number to quad precision (REAL(16)) type.

Syntax

```
result = QEXT (a)
```

a (Input) Must be of type integer, real, or complex.

Results

The result type is REAL(16) (REAL*16). Functions that cause conversion of one data type to another type have the same effect as the implied conversion in assignment statements.

If *a* is of type REAL(16), the result is the value of the *a* with no conversion (QEXT(a) = a).

If *a* is of type integer or real, the result has as much precision of the significant part of *a* as a REAL(16) value can contain.

If *a* is of type complex, the result has as much precision of the significant part of the real part of *a* as a REAL(16) value can contain.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(16)
	INTEGER(2)	REAL(16)
	INTEGER(4)	REAL(16)
	INTEGER(8)	REAL(16)
QEXT	REAL(4)	REAL(16)
QEXTD	REAL(8)	REAL(16)

Specific Name [1]	Argument Type	Result Type
	REAL(16)	REAL(16)
	COMPLEX(4)	REAL(16)
	COMPLEX(8)	REAL(16)
	COMPLEX(16)	REAL(16)

[1]These specific functions cannot be passed as actual arguments.

Example

QEXT (4) has the value 4.0 (rounded; there are 32 places to the right of the decimal point).

QEXT ((3.4, 2.0)) has the value 3.4 (rounded; there are 32 places to the right of the decimal point).

QFLOAT

Elemental Intrinsic Function (Generic):
Converts an integer to quad precision (REAL(16)) type.

Syntax

```
result = QFLOAT (a)
```

a (Input) Must be of type integer.

Results

The result type is REAL(16) (REAL*16).

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

Example

QFLOAT (-4) has the value -4.0 (rounded; there are 32 places to the right of the decimal point).

QNUM

Elemental Intrinsic Function (Specific): *Converts a character string to a REAL(16) value. This function cannot be passed as an actual argument.*

Syntax

```
result = QNUM (i)
```

i (Input) Must be of type character.

Results

The result type is REAL(16). The result value is the real value represented by the character string *i*.

Example

QNUM ("-174.23") has the value -174.23 of type REAL(16).

QRANSET

Portability Subroutine: *Sets the seed for a sequence of pseudo-random numbers.*

Module

USE IFPORT

Syntax

```
CALL QRANSET (rseed)
```

rseed (Input) INTEGER(4). The reset value for the seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

QREAL

Elemental Intrinsic Function (Specific): *Converts the real part of a COMPLEX(16) argument to REAL(16) type. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.*

Syntax

```
result = QREAL (a)
```

a	(Input) Must be of type COMPLEX(16) (or COMPLEX*32).

Results

The result type is quad-precision real (REAL(16) or REAL*16).

Example

QREAL ((2.0q0, 3.0q0)) has the value 2.0q0.

See Also

- Q to R
- REAL
- DREAL

QSORT

Portability Subroutine: *Performs a quick sort on an array of rank one.*

Module

USE IFPORT

Syntax

```
CALL QSORT (array,len,isize,compar)
```

array	(Input) Any type. One-dimensional array to be sorted. If the data type does not conform to one of the predefined interfaces for QSORT, you may have to create a new interface (see below).

len (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Number of elements in *array*.

isize (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Size, in bytes, of a single element of *array*:

- 4 if array is of type REAL(4)
- 8 if array is of type REAL(8) or complex
- 16 if array is of type COMPLEX(8)

compar (Input) INTEGER(2). Name of a user-defined ordering function that determines sort order. The type declaration of *compar* takes the form:
INTEGER(2) FUNCTION *compar*(*arg1*, *arg2*)
where *arg1* and *arg2* have the same type as *array* (above). Once you have created an ordering scheme, implement your sorting function so that it returns the following:

- Negative if arg1 should precede arg2
- Zero if arg1 is equivalent to arg2
- Positive if arg1 should follow arg2

Dummy argument *compar* must be declared as external.
In place of an INTEGER kind, you can specify the constant SIZEOF_SIZE_T, defined in IFPORT.F90, for argument *len* or *isize*. Use of this constant ensures correct compilation.

NOTE. If you use QSORT with different data types, your program must have a USE IFPORT statement so that all the calls work correctly. In addition, if you wish to use QSORT with a derived type or a type that is not in the predefined interfaces, you must include an overload for the generic subroutine QSORT. Examples of how to do this are in the portability module's source file, IFPORT.F90.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
PROGRAM SORTQ
   USE IFPORT
   integer(2), external :: cmp_function
   integer(2) insort(26), i
   integer (SIZEOF_SIZE_T) array_len, array_size
   array_len = 26
   array_size = 2
   do i=90,65,-1
     insort(i-64)=91 - i
   end do
   print *, "Before: "
   print *,insort
   CALL qsort(insort,array_len,array_size,cmp_function)
   print *, 'After: '
   print *, insort
END
!
   integer(2) function cmp_function(a1, a2)
   integer(2) a1, a2
   cmp_function=a1-a2
   end function
```

RADIX

Inquiry Intrinsic Function (Generic): *Returns the base of the model representing numbers of the same type and kind as the argument.*

Syntax

`result = RADIX (x)`

x (Input) Must be of type integer or real; it can be scalar or array valued.

Results

The result is a scalar of type default integer. For an integer argument, the result has the value r (as defined in Model for Integer Data). For a real argument, the result has the value b (as defined in Model for Real Data).

Example

If X is a REAL(4) value, RADIX (X) has the value 2.

See Also

- Q to R
- DIGITS
- EXPONENT
- FRACTION
- Data Representation Models

RAISEQQ

Portability Function: *Sends a signal to the executing program.*

Module

USE IFPORT

Syntax

`result = RAISEQQ (sig)`

sig (Input) INTEGER(4). Signal to raise. One of the following constants (defined in `IFPORT.F90`):

- SIG$ABORT - Abnormal termination
- SIG$FPE - Floating-point error
- SIG$ILL - Illegal instruction
- SIG$INT - CTRL+Csignal
- SIG$SEGV - Illegal storage access

- SIG$TERM - Termination request

If you do not install a signal handler (with SIGNALQQ, for example), when a signal occurs the system by default terminates the program with exit code 3.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero.

If a signal-handling routine for *sig* has been installed by a prior call to SIGNALQQ, RAISEQQ causes that routine to be executed. If no handler routine has been installed, the system terminates the program (the default action).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

See Also

- Q to R
- SIGNALQQ
- SIGNAL
- KILL

RAN

Nonelemental Intrinsic Function (Specific):
Returns the next number from a sequence of pseudorandom numbers of uniform distribution over the range 0 to 1. This is a specific function that has no generic function associated with it. It cannot be passed as an actual argument.

Syntax

```
result = RAN (i)
```

i	(Input; output) Must be an INTEGER(4) variable or array element. It should initially be set to a large, odd integer value. The RAN function stores a value in the argument that is later used to calculate the next random number. There are no restrictions on the seed, although it should be initialized with different values on separate runs to obtain different random numbers.

Results

The result type is REAL(4). The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. It is set equal to the value associated with the argument *i*.

RAN is not a pure function.

Example

In RAN (I), if variable I has the value 3, RAN has the value 4.8220158E-05.

See Also

- Q to R
- RANDOM
- RANDOM_NUMBER

RAND, RANDOM

Portability Functions: *Return real random numbers in the range 0.0 through 1.0.*

Module

USE IFPORT

Syntax

```
result = RAND ([ iflag])
result = RANDOM (iflag)
```

iflag	(Input) INTEGER(4). Optional for RAND. Controls the way the random number is selected.

Results

The result type is REAL(4). RAND and RANDOM return random numbers in the range 0.0 through 1.0.

Value of *iflag*	Selection process
1	The generator is restarted and the first random value is selected.

Value of *iflag*	Selection process
0	The next random number in the sequence is selected.
Otherwise	The generator is reseeded using *iflag*, restarted, and the first random value is selected.

When RAND is called without an argument, *iflag* is assumed to be 0.

There is no difference between RAND and RANDOM. Both functions are included to ensure portability of existing code that references one or both of them. The intrinsic functions RANDOM_NUMBER and RANDOM_SEED provide the same functionality.

You can use SRAND to restart the pseudorandom number generator used by RAND.

NOTE. RANDOM is available as a function or subroutine.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

The following example shows how to use both the RANDOM function and the RANDOM subroutine:

```
use ifport real(4) ranval

!from libifcore.lib

call seed(1995) ! initialize

!also from for_m_irand.c in libfor

call random(ranval) ! get next random number

print *,ranval
```

```
!from libifport.lib

ranval = random(1) ! initialize

! same

ranval = random(0) ! get next random number

print *,ranval

end
```

See Also

- Q to R
- RANDOM_NUMBER
- RANDOM_SEED
- SRAND

RANDOM Subroutine

Portability Subroutine: *Returns a pseudorandom number greater than or equal to zero and less than one from the uniform distribution.*

Module

USE IFPORT

Syntax

`CALL RANDOM (`*`ranval`*`)`

`ranval`	(Output) REAL(4). Pseudorandom number, 0 *`ranval`*< 1, from the uniform distribution.

A given seed always produces the same sequence of values from RANDOM.

If SEED is not called before the first call to RANDOM, RANDOM begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in `IFCORE.F90`) to SEED before the first call to RANDOM.

The portability routines DRAND, DRANDM, IRAND, IRANDM, RAN, RAND, and the RANDOM portability function and subroutine use the same algorithms and thus return the same answers. They are all compatible and can be used interchangeably. The algorithm used is a "Prime Modulus M Multiplicative Linear Congruential Generator," a modified version of the random number generator by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find," CACM, October 1988, Vol. 31, No. 10.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
REAL(4) ran
CALL SEED(1995)
CALL RANDOM(ran)
```

The following example shows how to use both the RANDOM subroutine and the RANDOM function:

```
use ifport
real(4) ranval
!from libifcore.lib
call seed(1995)     ! initialize
!also from for_m_irand.c in libfor
call random(ranval) ! get next random number
print *,ranval
!from libifport.lib
ranval = random(1)  ! initialize
! same
ranval = random(0)  ! get next random number
print *,ranval
end
```

See Also

- Q to R
- RANDOM_NUMBER
- SEED
- DRAND and DRANDM
- IRAND and IRANDM
- RAN
- RAND

RANDOM_NUMBER

Intrinsic Subroutine: *Returns one pseudorandom number or an array of such numbers.*

Syntax

```
CALL RANDOM_NUMBER (harvest)
```

harvest	(Output) Must be of type real. It can be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution within the range 0 <= *x* < 1.

The seed for the pseudorandom number generator used by RANDOM_NUMBER can be set or queried with RANDOM_SEED. If RANDOM_SEED is not used, the processor sets the seed for RANDOM_NUMBER to a processor-dependent value.

The RANDOM_NUMBER generator uses two separate congruential generators together to produce a period of approximately 10**18, and produces real pseudorandom results with a uniform distribution in (0,1). It accepts two integer seeds, the first of which is reduced to the range [1, 2147483562]. The second seed is reduced to the range [1, 2147483398]. This means that the generator effectively uses two 31-bit seeds.

For more information on the algorithm, see the following:

- Communications of the ACM vol 31 num 6 June 1988, titled: Efficient and Portable Combined Random Number Generators by Pierre L'ecuyer.
- Springer-Verlag New York, N. Y. 2nd ed. 1987, titled: A Guide to Simulation by Bratley, P., Fox, B. L., and Schrage, L. E.

Example

Consider the following:

```
REAL Y, Z (5, 5)
! Initialize Y with a pseudorandom number
CALL RANDOM_NUMBER (HARVEST = Y)
CALL RANDOM_NUMBER (Z)
```

Y and Z contain uniformly distributed random numbers.

The following shows another example:

```
 REAL x, array1 (5, 5)
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(x)
 CALL RANDOM_NUMBER(array1)
```

Consider also the following:

```
program testrand
  intrinsic random_seed, random_number
  integer size, seed(2), gseed(2), hiseed(2), zseed(2)
  real harvest(10)
  data seed /123456789, 987654321/
  data hiseed /-1, -1/
  data zseed /0, 0/
  call random_seed(SIZE=size)
  print *,"size ",size
  call random_seed(PUT=hiseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"hiseed gseed", hiseed, gseed
  call random_seed(PUT=zseed(1:size))
  call random_seed(GET=gseed(1:size))
  print *,"zseed gseed ", zseed, gseed
  call random_seed(PUT=seed(1:size))
  call random_seed(GET=gseed(1:size))
  call random_number(HARVEST=harvest)
  print *, "seed gseed ", seed, gseed
  print *, "harvest"
  print *, harvest
  call random_seed(GET=gseed(1:size))
  print *,"gseed after harvest ", gseed
end program testrand
```

See Also

- Q to R
- RANDOM_SEED
- RANDOM

- SEED
- DRAND and DRANDM
- IRAND and IRANDM
- RAN
- RAND and RANDOM

RANDOM_SEED

Intrinsic Subroutine (Generic): *Changes or queries the seed (starting point) for the pseudorandom number generator used by RANDOM_NUMBER. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL RANDOM_SEED ([size] [,put] [,get])
```

size	(Output; optional) Must be scalar and of type integer. Set to the number of integers (N) that the processor uses to hold the value of the seed.
put	(Input; optional) Must be an integer array of rank one and size greater than or equal to N. It is used to reset the value of the seed.
get	(Output; optional) Must be an integer array of rank one and size greater than or equal to N. It is set to the current value of the seed.

No more than one argument can be specified. If no argument is specified, a random number based on the date and time is assigned to the seed.

At run-time, the arguments are examined in the order *size* then *put* then *get*. The first optional argument in this order that is present determines the behavior of the RANDOM_SEED call.

You can determine the size of the array the processor uses to store the seed by calling RANDOM_SEED with the *size* argument (see the second example below).

Example

Consider the following:

```
CALL RANDOM_SEED                        ! Processor initializes the
                                        !   seed randomly from the date
                                        !   and time
CALL RANDOM_SEED (SIZE = M)             ! Sets M to N
CALL RANDOM_SEED (PUT = SEED (1 : M))   ! Sets user seed
CALL RANDOM_SEED (GET = OLD  (1 : M))   ! Reads current seed
```

The following shows another example:

```
 INTEGER I
 INTEGER, ALLOCATABLE :: new (:), old(:)
 CALL RANDOM_SEED ( )  ! Processor reinitializes the seed
                       ! randomly from the date and time
 CALL RANDOM_SEED (SIZE = I)  ! I is set to the size of
                              ! the seed array
 ALLOCATE (new(I))
 ALLOCATE (old(I))
 CALL RANDOM_SEED (GET=old(1:I)) ! Gets the current seed
 WRITE(*,*) old
 new = 5
 CALL RANDOM_SEED (PUT=new(1:I)) ! Sets seed from array
                                 ! new
 END
```

See Also

- Q to R
- RANDOM_NUMBER
- SEED
- SRAND

RANDU

Intrinsic Subroutine (Generic): *Computes a pseudorandom number as a single-precision value. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

```
CALL RANDU (i1,i2,x)
```

i1, i2	(Input; output) Must be scalars of type INTEGER(2) or INTEGER(4). They contain the *seed* for computing the random number. These values are updated during the computation so that they contain the updated seed.
x	(Output) Must be a scalar of type REAL(4). This is where the computed random number is returned.

The result is returned in *x*, which must be of type REAL(4). The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for *i1* and *i2*.

The result value is a pseudorandom number in the range 0.0 to 1.0. The algorithm for computing the random number value is based on the values for *i1* and *i2*.

If *i1* = 0 and *i2* = 0, the generator base is set as follows:

```
x(n + 1) = 2**16 + 3
```

Otherwise, it is set as follows:

```
x(n + 1) = (2**16 + 3) * x(n) mod 2**32
```

The generator base x(n + 1) is stored in *i1*, *i2*. The result is x(n + 1) scaled to a real value y(n + 1), for 0.0 <= y(n + 1) < 1.

Example

Consider the following:

```
REAL X

INTEGER(2) I, J

...

CALL RANDU (I, J, X)
```

If I and J are values 4 and 6, X has the value 5.4932479E-04.

RANF

Portability Function: *Generates a random number between 0.0 and RAND_MAX.*

Module

USE IFPORT

Syntax

```
result = RANF ( )
```

Results

The result type is REAL(4). The result value is a single-precision pseudo-random number between 0.0 and RAND_MAX as defined in the C library, normally 0x7FFF 215-1.

The initial seed is set by the following:

```
CALL SRAND(ISEED)
```

where ISEED is type INTEGER(4).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

RANGE

Inquiry Intrinsic Function (Generic): *Returns the decimal exponent range in the model representing numbers with the same kind parameter as the argument.*

Syntax

```
result = RANGE (x)
```

x	(Input) Must be of type integer, real, or complex; it can be scalar or array valued.

Results

The result is a scalar of type default integer.

For an integer argument, the result has the value INT(LOG10(HUGE(*x*))). For information on the integer model, see Model for Integer Data.

For a real or complex argument, the result has the value INT(MIN (LOG10(HUGE(*x*)), -LOG10(TINY(*x*)))). For information on the real model, see Model for Real Data.

Example

If X is a REAL(4) value, RANGE (X) has the value 37. (HUGE(X) = (1 - 2^{-24}) x $2^{128}$and TINY(X) = 2^{-126})

See Also

- Q to R
- HUGE
- TINY

RANGET

Portability Subroutine: *Returns the current seed.*

Module

USE IFPORT

Syntax

`CALL RANGET (seed)`

seed	(Output) INTEGER(4). The current seed value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

RANSET

Portability Subroutine: *Sets the seed for the random number generator.*

Module

USE IFPORT

Syntax

`CALL RANSET (seed)`

seed	(Input) REAL(4). The reset value for the seed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

READ Statement

Statement: *Transfers input data from external sequential, direct-access, or internal records.*

Syntax

Sequential

Formatted:

```
READ (eunit, format [, advance] [, asynchronous] [, id] [, pos] [, size] [, iostat] [, err] [, end] [, eor]) [io-list]
```

```
READ form[, io-list]
```

Formatted - List-Directed:

```
READ (eunit, *[, asynchronous] [, id] [, pos] [, iostat] [, err] [, end]) [io-list]
```

```
READ *[, io-list]
```

Formatted - Namelist:

```
READ (eunit, nml-group[, iostat][, err][, end])
```

```
READ nml
```

Unformatted:

```
READ (eunit [, asynchronous] [, id] [, pos] [, iostat][, err][, end]) [io-list]
```

Direct-Access
Formatted:

```
READ (eunit, format, rec [, asynchronous] [, id] [, pos] [, iostat] [, err]) [io-list]
```

Unformatted:

```
READ (eunit, rec [, asynchronous] [, id] [, pos]  [, iostat] [, err]) [io-list]
```

Internal

```
READ (iunit, format [, iostat] [, err] [, end]) [io-list]
```

eunit	Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.
format	Is a format specifier. It is optionally prefaced by FMT= if *format* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=. For internal READs, an asterisk (*) indicates list-directed formatting. For direct-access READs, an asterisk is not permitted.
advance	Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.
asynchronous	Is an asynchronous specifier (ASYNCHRONOUS=*i-expr*). If the value of *i-expr* is 'YES', the statement uses asynchronous input; if the value is 'NO', the statement uses synchronous input. The default value is 'NO'.
id	Is an id specifier (ID=*id-var*). If ASYNCHRONOUS='YES' is specified and the operation completes successfully, the id specifier becomes defined with an implementation-dependent value that can be specified in a future WAIT or INQUIRE statement to identify the particular data transfer operation. If an error occurs, the id specifier variable becomes undefined.
pos	Is a pos specifier (POS=*p*) that indicates a file position in file storage units in a stream file (ACCESS='STREAM'). It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.
size	Is a character count specifier (SIZE=*i-var*). It can only be specified for nonadvancing READ statements.
iostat	Is the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.
err, end, eor	Are branch specifiers if an error (ERR=*label*), end-of-file (END=*label*), or end-of-record (EOR=*label*) condition occurs. EOR can only be specified for nonadvancing READ statements.
io-list	Is an I/O list: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-DO list.

	If an item in *io-list* is an expression that calls a function, that function must not execute an I/O statement or the EOF intrinsic function on the same external unit as *eunit*. If I/O is to or from a formatted device, *io-list* cannot contain derived-type variables, but it can contain components of derived types. If I/O is to a binary or unformatted device, *io-list* can contain either derived type components or a derived type variable.
form	Is the nonkeyword form of a format specifier (no FMT=).
*	Is the format specifier indicating list-directed formatting. (It can also be specified as FMT=*.)
nml-group	Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier. For more information, see Namelist Specifier.
nml	Is the nonkeyword form of a namelist specifier (no NML=) indicating namelist I/O.
rec	Is the cell number of a record to be accessed directly. Optionally prefaced by REC=.
iunit	Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list. It must be a character variable. It must not be an array section with a vector subscript.

CAUTION. The READ statement can disrupt the results of certain graphics text functions (such as SETTEXTWINDOW) that alter the location of the cursor. You can avoid the problem by getting keyboard input with the GETCHARQQ function and echoing the keystrokes to the screen using OUTTEXT. Alternatively, you can use SETTEXTPOSITION to control cursor location.

Example

```
DIMENSION ia(10,20)
! Read in the bounds for the array.
! Then read in the array in nested implied-DO lists
! with input format of 8 columns of width 5 each.
READ (6, 990) il, jl, ((ia(i,j), j = 1, jl), i =1, il)
990 FORMAT (2I5, /, (8I5))
! Internal read gives a variable string-represented numbers
CHARACTER*12 str
str = '123456'
READ (str,'(i6)') i
! List-directed read uses no specified format
REAL x, y
INTEGER i, j
READ (*,*) x, y, i, j
```

See Also

- Q to R
- I/O Lists
- I/O Control List
- Forms for Sequential READ Statements
- Forms for Direct-Access READ Statements
- Forms and Rules for Internal READ Statements
- PRINT
- WRITE
- I/O Formatting

REAL Statement

Statement: *Specifies the REAL data type.*

Syntax

```
REAL
```

```
REAL([KIND=] n)
REAL* n
DOUBLE PRECISION
```

n Is an initialization expression that evaluates to kind 4, 8 or 16.

Description

If a kind parameter is specified, the real constant has the kind specified. If a kind parameter is not specified, the kind is default real.

Default real is affected by compiler options specifying real size and by the REAL directive.

The default KIND for DOUBLE PRECISION is affected by compiler option double-size. If this compiler option is not specified, default DOUBLE PRECISION is REAL(8).

No kind parameter is permitted for data declared with type DOUBLE PRECISION.

REAL(4) and REAL*4 (single precision) are the same data type. REAL(8), REAL*8, and DOUBLE PRECISION are the same data type.

Example

Entity-oriented examples are:

```
MODULE DATDECLARE
   REAL (8), OPTIONAL :: testval=50.d0
   REAL, SAVE :: a(10), b(20,30)
   REAL, PARAMETER :: x = 100.
Attribute-oriented examples are:
MODULE DATDECLARE
   REAL (8) testval=50.d0
   REAL x, a(10), b(20,30)
   OPTIONAL testval
   SAVE a, b
   PARAMETER (x = 100.)
```

See Also

- Q to R

- DOUBLE PRECISION
- REAL directive
- Real Data Types
- General Rules for Real Constants
- REAL(4) Constants
- REAL(8) or DOUBLE PRECISION Constants
- Real and Complex Editing
- Model for Real Data

REAL Directive

General Compiler Directive: *Specifies the default real kind.*

Syntax

```
cDEC$ REAL:{ 4 | 8 | 16 }
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The REAL directive selects a size of 4 (KIND=4), 8 (KIND=8), or 16 (KIND=16) bytes for default real numbers. When the directive is in effect, all default real and complex variables are of the kind specified in the directive. Only numbers specified or implied as REAL without KIND are affected.

The REAL directive can appear only at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. REAL cannot appear between program units, or at the beginning of internal subprograms. It does not affect modules invoked with the USE statement in the program unit that contains it.

Example

```
REAL r              ! a 4-byte REAL
WRITE(*,*) KIND(r)
CALL REAL8( )
WRITE(*,*) KIND(r)  ! still a 4-byte REAL
                    !   not affected by setting in subroutine
END
SUBROUTINE REAL8( )
  !DEC$ REAL:8
  REAL s ! an 8-byte REAL
  WRITE(*,*) KIND(s)
END SUBROUTINE
```

See Also

- Q to R
- REAL
- COMPLEX
- General Compiler Directives

REAL Function

Elemental Intrinsic Function (Generic):
Converts a value to real type.

Syntax

`result = REAL (a[,kind])`

a	(Input) Must be of type integer, real, or complex.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is real. If *kind* is present, the kind parameter is that specified by *kind*; otherwise, the kind parameter of the result is shown in the following table. If the processor cannot represent the result value in the kind of the result, the result is undefined.

Functions that cause conversion of one data type to another type have the same affect as the implied conversion in assignment statements.

If *a* is integer or real, the result is equal to an approximation of *a*. If *a* is complex, the result is equal to an approximation of the real part of *a*.

Specific Name [1]	Argument Type	Result Type
	INTEGER(1)	REAL(4)
FLOATI	INTEGER(2)	REAL(4)
FLOAT [2, 3]	INTEGER(4)	REAL(4)
REAL [2]	INTEGER(4)	REAL(4)
FLOATK	INTEGER(8)	REAL(4)
	REAL(4)	REAL(4)
SNGL [2, 4]	REAL(8)	REAL(4)
SNGLQ	REAL(16)	REAL(4)
	COMPLEX(4)	REAL(4)
	COMPLEX(8)	REAL(8)

[1] These specific functions cannot be passed as actual arguments.

[2] The setting of compiler options specifying real size can affect FLOAT, REAL, and SNGL.

[3] Or FLOATJ. For compatibility with older versions of Fortran, FLOAT is generic, allowing any kind of INTEGER argument, and returning a default real result.

[4] For compatibility with older versions of Fortran, SNGL is generic, allowing any kind of REAL argument, and returning a default real result.

Example

REAL (-4) has the value -4.0.

REAL (Y) has the same kind parameter and value as the real part of complex variable Y.

See Also

- Q to R
- DFLOAT
- DREAL
- DBLE

RECORD

Statement: *Declares a record structure as an entity with a name.*

Syntax

```
RECORD /structure-name/record-namelist
   /structure-name/record-namelist]
   . . .
   [, /structure-name/record-namelist]
```

structure-name	Is the name of a previously declared structure.
record-namelist	Is a list of one or more variable names, array names, or array specifications, separated by commas. All of the records named in this list have the same structure and are allocated separately in memory.

You can use record names in COMMON and DIMENSION statements, but not in DATA or NAMELIST statements.

Records initially have undefined values unless you have defined their values in structure declarations.

STRUCTURE and RECORD constructs have been replaced by derived types, which should be used in writing new code. See Derived Data Types.

Example

```
STRUCTURE /address/
  LOGICAL*2    house_or_apt
  INTEGER*2    apt
  INTEGER*2    housenumber
  CHARACTER*30 street
  CHARACTER*20 city
  CHARACTER*2  state
  INTEGER*4    zip
END STRUCTURE
RECORD /address/ mailing_addr(20), shipping_addr(20)
```

See Also

- Q to R
- TYPE
- MAP...END MAP
- STRUCTURE...END STRUCTURE
- UNION...END UNION
- Record Structures

RECTANGLE, RECTANGLE_W (W*32, W*64)

Graphics Functions: *Draw a rectangle using the current graphics color, logical write mode, and line style.*

Module

USE IFQWIN

Syntax

```
result = RECTANGLE (control, x1, y1, x2, y2)
result = RECTANGLE_W (control, wx1, wy1, wx2, wy2)
```

control	(Input) INTEGER(2). Fill flag. One of the following symbolic constants (defined in IFQWIN.F90): • $GFILLINTERIOR - Draws a solid figure using the current color and fill mask. • $GBORDER - Draws the border of a rectangle using the current color and line style.
x1, y1	(Input) INTEGER(2). Viewport coordinates for upper-left corner of rectangle.
x2, y2	(Input) INTEGER(2). Viewport coordinates for lower-right corner of rectangle.
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of rectangle.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of rectangle.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0.

The RECTANGLE function uses the viewport-coordinate system. The viewport coordinates (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle.

The RECTANGLE_W function uses the window-coordinate system. The window coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*) are the diagonally opposed corners of the rectangle.

SETCOLORRGB sets the current graphics color. SETFILLMASK sets the current fill mask. By default, filled graphic shapes are filled solid with the current color.

If you fill the rectangle using FLOODFILLRGB, the rectangle must be bordered by a solid line style. Line style is solid by default and can be changed with SETLINESTYLE.

NOTE. The RECTANGLE routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the Rectangle routine by including the IFWIN module, you need to specify the routine name as MSFWIN$Rectangle. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

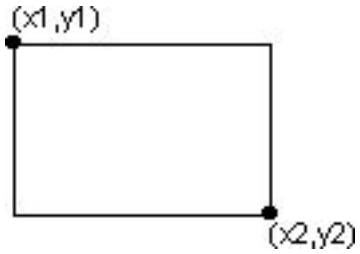
Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

This program draws the rectangle shown below.

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) dummy, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
dummy = RECTANGLE( $GBORDER, x1, y1, x2, y2 )
END
```



See Also

- Q to R
- SETFILLMASK
- GRSTATUS
- LINETO
- POLYGON
- FLOODFILLRGB
- SETLINESTYLE
- SETCOLOR
- SETWRITEMODE

Building Applications: Drawing Lines on the Screen

Building Applications: Graphics Coordinates

RECURSIVE

Keyword: *Specifies that a subroutine or function can call itself directly or indirectly. Recursion is permitted if the keyword is specified in a FUNCTION or SUBROUTINE statement, or if RECURSIVE is specified as a compiler option or in an OPTIONS statement.*

Description

If a function is directly recursive and array valued, the keywords RECURSIVE and RESULT must *both* be specified in the FUNCTION statement.

The procedure interface is explicit within the subprogram in the following cases:

- When RECURSIVE is specified for a subroutine
- When RECURSIVE and RESULT are specified for a function

The keyword RECURSIVE *must* be specified if any of the following applies (directly or indirectly):

- The subprogram invokes itself.
- The subprogram invokes a subprogram defined by an ENTRY statement in the same subprogram.
- An ENTRY procedure in the same subprogram invokes one of the following:
  - Itself
  - Another ENTRY procedure in the same subprogram
  - The subprogram defined by the FUNCTION or SUBROUTINE statement

Example

```
! RECURS.F90
!
  i = 0
  CALL Inc (i)
  END
  RECURSIVE SUBROUTINE Inc (i)
  i = i + 1
  CALL Out (i)
  IF (i.LT.20) CALL Inc (i)    ! This also works in OUT
  END SUBROUTINE Inc
  SUBROUTINE Out (i)
  WRITE (*,*) i
  END SUBROUTINE Out
```

See Also

- Q to R
- ENTRY
- FUNCTION
- SUBROUTINE
- OPTIONS
- Program Units and Procedures
- recursive compiler option

REDUCTION

Parallel Directive Clause: *Performs a reduction operation on the specified variables.*

Syntax

REDUCTION (*operator*| *intrinsic*: *list*)

operator	Is one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV.
intrinsic	Is one of the following: MAX, MIN, IAND, IOR, or IEOR.

list	Is the name of one or more variables of intrinsic type that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma. Deferred-shape and assumed-size arrays are not allowed.

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the operator (see the tablebelow).

At the end of the REDUCTION, the shared variable is updated to reflect the result of combining the original value of the shared reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value; the partial results of a subtraction reduction are added to form the final value.

The value of the shared variable becomes undefined when the first thread reaches the clause containing the reduction, and it remains undefined until the reduction computation is complete. Normally, the computation is complete at the end of the REDUCTION construct.

However, if the REDUCTION clause is used in a construct to which NOWAIT is also applied, the shared variable remains undefined until a barrier synchronization has been performed. This ensures that all the threads complete the REDUCTION clause.

The REDUCTION clause must be used in a region or worksharing construct where the reduction variable is used only in a reduction statement having one of the following forms:

```
x = x operator expr

x = expr operator x (except for subtraction)

x = intrinsic (x, expr)

x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a MAX reduction can be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. Be careful that the operator you specify in the REDUCTION clause matches the reduction operation.

The following table lists the operators and intrinsics and their initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 899: Initialization Values for REDUCTION Operators

Operator	Initialization Value
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.

Table 900: Initialization Values for REDUCTION Intrinsics

Intrinsic	Initialization Value
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

If a directive allows reduction clauses, the number you can specify is not limited. However, each variable name can appear in only one of the clauses.

See Also

- Q to R

Optimizing Applications: OpenMP* Directives and Clauses Summary

Optimizing Applications: REDUCTION Clause

Optimizing Applications: Data Scope Attribute Clauses Overview

Optimizing Applications: Parallel Region Directives

%REF

Built-in Function: *Changes the form of an actual argument. Passes the argument by reference. In Intel® Fortran, passing by reference is the default.*

Syntax

```
%REF (a)
```

a	(Input) An expression, record name, procedure name, array, character array section, or array element.

You must specify %REF in the actual argument list of a CALL statement or function reference. You cannot use it in any other context.

The following table lists the Intel Fortran defaults for argument passing, and the allowed uses of %REF:

Actual Argument Data Type	Default	%REF
Expressions:		
Logical	REF	Yes
Integer	REF	Yes
REAL(4)	REF	Yes
REAL(8)	REF	Yes
REAL(16)	REF	Yes
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
COMPLEX(16)	REF	Yes
Character	See table note [1]	Yes
Hollerith	REF	No

Actual Argument Data Type	Default	%REF
Aggregate[2]	REF	Yes
Derived	REF	Yes
Array Name:		
Numeric	REF	Yes
Character	See table note [1]	Yes
Aggregate[2]	REF	Yes
Derived	REF	Yes
Procedure Name:		
Numeric	REF	Yes
Character	See table note [1]	Yes

[1]A character argument is passed by address and hidden length.

[2]In Intel Fortran record structures

The %REF and %VAL functions override related cDEC$ ATTRIBUTE settings.

Example

```
CHARACTER(LEN=10) A, B
CALL SUB(A, %REF(B))
```

Variable A is passed by address and hidden length. Variable B is passed by reference.

Note that on Windows systems, compiler option iface determines how the character argument for variable B is passed.

See Also

- Q to R
- CALL
- %VAL

- %LOC
- /iface compiler option

REGISTERMOUSEEVENT (W*32, W*64)

QuickWin Function: *Registers the application-supplied callback routine to be called when a specified mouse event occurs in a specified window.*

Module

USE IFQWIN

Syntax

`result = REGISTERMOUSEEVENT (unit,mouseevents,callbackroutine)`

unit	(Input) INTEGER(4). Unit number of the window whose callback routine on mouse events is to be registered.
mouseevents	(Input) INTEGER(4). One or more mouse events to be handled by the callback routine to be registered. Symbolic constants (defined in `IFQWIN.F90`) for the possible mouse events are: • MOUSE$LBUTTONDOWN - Left mouse button down • MOUSE$LBUTTONUP - Left mouse button up • MOUSE$LBUTTONDBLCLK - Left mouse button double-click • MOUSE$RBUTTONDOWN - Right mouse button down • MOUSE$RBUTTONUP - Right mouse button up • MOUSE$RBUTTONDBLCLK - Right mouse button double-click • MOUSE$MOVE - Mouse moved
callbackroutine	(Input) Routine to be called on the specified mouse event in the specified window. It must be declared EXTERNAL. For a prototype mouse callback routine, see *Building Applications*: Using a Mouse.

Results

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

- MOUSE$BADUNIT - The unit specified is not open, or is not associated with a QuickWin window.
- MOUSE$BADEVENT - The event specified is not supported.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Compatibility

QUICKWIN GRAPHICS LIB

Example

The following example registers the routine CALCULATE, to be called when the user double-clicks the left mouse button while the mouse cursor is in the child window opened as unit 4:

```
USE IFQWIN

INTEGER(4) result

OPEN (4, FILE= 'USER')

...

result = REGISTERMOUSEEVENT (4, MOUSE$LBUTTONDBLCLK, CALCULATE)
```

See Also

- Q to R
- UNREGISTERMOUSEEVENT
- WAITONMOUSEEVENT

REMAPALLPALETTERGB, REMAPPALETTERGB (W*32, W*64)

Graphics Functions: *REMAPALLPALETTERGB remaps a set of Red-Green-Blue (RGB) color values to indexes recognized by the video hardware. REMAPPALETTERGB remaps one color index to an RGB color value.*

Module

USE IFQWIN

Syntax

```
result = REMAPALLPALETTERGB (colors)
result = REMAPPALETTERGB (index, colors)
```

colors	(Input) INTEGER(4). Ordered array of RGB color values to be mapped in order to indexes. Must hold 0-255 elements.
index	(Input) INTEGER(4). Color index to be reassigned an RGB color.
color	(Input) INTEGER(4). RGB color value to assign to a color index.

Results

The result type is INTEGER(4). REMAPALLPALETTERGB returns 0 if successful; otherwise, -1. REMAPPALETTERGB returns the previous color assigned to the index.

The REMAPALLPALETTERGB function remaps all of the available color indexes simultaneously (up to 236; 20 indexes are reserved by the operating system). The *colors* argument points to an array of RGB color values. The default mapping between the first 16 indexes and color values is shown in the following table. The 16 default colors are provided with symbolic constants in IFQWIN.F90.

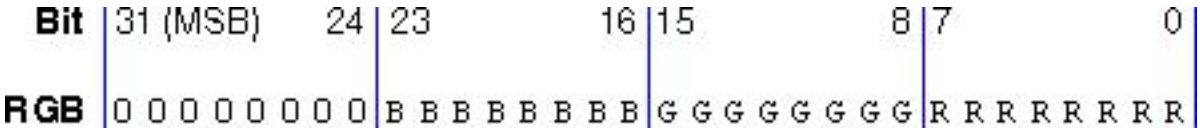
Index	Color	Index	Color
0	$BLACK	8	$GRAY
1	$BLUE	9	$LIGHTBLUE
2	$GREEN	10	$LIGHTGREEN
3	$CYAN	11	$LIGHTCYAN
4	$RED	12	$LIGHTRED
5	$MAGENTA	13	$LIGHTMAGENTA
6	$BROWN	14	$YELLOW
7	$WHITE	15	$BRIGHTWHITE

The number of colors mapped can be fewer than 236 if the number of colors supported by the current video mode is fewer, but at most 236 colors can be mapped by REMAPALLPALETTERGB. Most Windows graphics drivers support a palette of 256K colors or more, of which only a few

can be mapped into the 236 palette indexes at a time. To access and use all colors on the system, bypass the palette and use direct RGB color functions such as such as SETCOLORRGB and SETPIXELSRGB.

Any RGB colors can be mapped into the 236 palette indexes. Thus, you could specify a palette with 236 shades of red. For further details on using different color procedures see *Building Applications*: Adding Color Overview.

In each RGB color value, each of the three colors, red, green and blue, is represented by an eight-bit value (2 hex digits). In the values you specify with REMAPALLPALETTERGB or REMAPPALETTERGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 11111111 (hex FF) the maximum for each of the three components. For example, Z'008080' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(4) colors(3)
INTEGER(2) status
colors(1) = Z'00FFFF' ! yellow
colors(2) = Z'FFFFFF' ! bright white
colors(3) = 0       ! black
status = REMAPALLPALETTERGB(colors)
status = REMAPPALETTERGB(INT2(47), Z'45A315')
END
```

See Also

- Q to R
- SETBKCOLORRGB
- SETCOLORRGB
- SETBKCOLOR
- SETCOLOR

Building Applications: Using Color

Building Applications: VGA Color Palette

RENAME

Portability Function: *Renames a file.*

Module

USE IFPORT

Syntax

`result = RENAME (from,to)`

from	(Input) Character*(*). Path of an existing file.
to	(Input) Character*(*). The new path for the file (see Caution note below).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code, such as:

- EACCES - The file or directory specified by *to* could not be created (invalid path). This error is also returned if the drive specified is not currently connected to a device.
- ENOENT - The file or path specified by *from* could not be found.
- EXDEV - Attempt to move a file to a different device.

CAUTION. This routine can cause data to be lost. If the file specified in *to* already exists, RENAME deletes the pre-existing file.

It is possible to rename a file to itself without error.

The paths can use forward (/) or backward (\) slashes as path separators and can include drive letters (if permitted by your operating system).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
use IFPORT
integer(4) istatus
character*12 old_name, new_name
print *, "Enter file to rename: "
read *, old_name
print *, "Enter new name: "
read *, new_name
ISTATUS = RENAME (old_name, new_name)
```

See Also

- Q to R
- RENAMEFILEQQ

RENAMEFILEQQ

Portability Function: *Renames a file.*

Module

USE IFPORT

Syntax

`result = RENAMEFILEQQ (oldname,newname)`

oldname	(Input) Character*(*). Current name of the file to be renamed.
newname	(Input) Character*(*). New name of the file to be renamed.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

You can use RENAMEFILEQQ to move a file from one directory to another on the same drive by giving a different path in the *newname* parameter.

If the function fails, call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

- ERR$ACCES - Permission denied. The file's permission setting does not allow the specified access.
- ERR$EXIST - The file already exists.
- ERR$NOENT - File or path specified by *oldname* not found.
- ERR$XDEV - Attempt to move a file to a different device.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
USE IFCORE
INTEGER(4) len
CHARACTER(80) oldname, newname
LOGICAL(4) result
WRITE(*,'(A, \)') ' Enter old name: '
len = GETSTRQQ(oldname)
WRITE(*,'(A, \)') ' Enter new name: '
len = GETSTRQQ(newname)
result = RENAMEFILEQQ(oldname, newname)
END
```

See Also

- Q to R
- FINDFILEQQ
- RENAME
- GETLASTERRORQQ

REPEAT

Transformational Intrinsic Function (Generic):
Concatenates several copies of a string.

Syntax

```
result = REPEAT (string,ncopies)
```

string	(Input) Must be scalar and of type character.
ncopies	(Input) Must be scalar and of type integer. It must not be negative.

Results

The result is a scalar of type character and length *ncopies* x LEN(*string*). The kind parameter is the same as *string*. The value of the result is the concatenation of *ncopies* copies of *string*.

Example

REPEAT ('S', 3) has the value SSS.

REPEAT ('ABC', 0) has the value of a zero-length string.

The following shows another example:

```
CHARACTER(6) str

str = REPEAT('HO', 3)  ! returns HOHOHO
```

See Also

- Q to R
- SPREAD

RESHAPE

Transformational Intrinsic Function (Generic):
Constructs an array with a different shape from the argument array.

Syntax

```
result = RESHAPE (source,shape[,pad] [,order])
```

source	(Input) Must be an array. It may be of any data type. It supplies the elements for the result array. Its size must be greater than or equal to PRODUCT(*shape*) if *pad* is omitted or has size zero.

shape	(Input) Must be an integer array of up to 7 elements, with rank one and constant size. It defines the shape of the result array. Its size must be positive; its elements must not have negative values.
pad	(Input; optional) Must be an array with the same type and kind parameters as *source*. It is used to fill in extra values if the result array is larger than *source*.
order	(Input; optional) Must be an integer array with the same shape as *shape*. Its elements must be a permutation of (1,2,...,n), where n is the size of *shape*. If *order* is omitted, it is assumed to be (1,2,...,n).

Results

The result is an array of shape *shape* with the same type and kind parameters as *source*. The size of the result is the product of the values of the elements of *shape*.

In the result array, the array elements of *source* are placed in the order of dimensions specified by *order*. If *order* is omitted, the array elements are placed in normal array element order.

The array elements of *source* are followed (if necessary) by the array elements of *pad* in array element order. If necessary, additional copies of *pad* follow until all the elements of the result array have values.

NOTE. In standard Fortran array element order, the first dimension varies fastest. For example, element order in a two-dimensional array would be (1,1), (2,1), (3,1) and so on. In a three-dimensional array, each dimension having two elements, the array element order would be (1,1,1), (2, 1, 1), (1, 2, 1), (2, 2, 1), (1, 1, 2), (2, 1, 2), (1, 2, 2), (2, 2, 2).

RESHAPE can be used to reorder a Fortran array to match C array ordering before the array is passed from a Fortran to a C procedure.

Example

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 3/)) has the value

```
[ 3  5  7 ]
[ 4  6  8 ].
```

RESHAPE ((/3, 4, 5, 6, 7, 8/), (/2, 4/), (/1, 1/), (/2, 1/)) has the value

```
[ 3  4  5  6 ]
[ 7  8  1  1 ].
```

The following shows another example:

```
INTEGER AR1( 2, 5)
REAL F(5,3,8)
REAL C(8,3,5)
AR1 = RESHAPE((/1,2,3,4,5,6/),(/2,5/),(/0,0/),(/2,1/))
! returns      1 2 3 4 5
!              6 0 0 0 0
!
! Change Fortran array order to C array order
C = RESHAPE(F, (/8,3,5/), ORDER = (/3, 2, 1/))
END
```

See Also

- Q to R
- PACK
- SHAPE
- TRANSPOSE
- Array Assignment Statements

RESULT

Keyword: *Specifies a name for a function result.*

Description

Normally, a function result is returned in the function's name, and all references to the function name are references to the function result.

However, if you use the RESULT keyword in a FUNCTION statement, you can specify a local variable name for the function result. In this case, all references to the function name are recursive calls, and the function name must not appear in specification statements.

The RESULT name must be different from the name of the function.

Example

The following shows an example of a recursive function specifying a RESULT variable:

```
RECURSIVE FUNCTION FACTORIAL(P) RESULT(L)
  INTEGER, INTENT(IN) :: P
  INTEGER L
  IF (P == 1) THEN
    L = 1
  ELSE
    L = P * FACTORIAL(P - 1)
  END IF
END FUNCTION
```

The following shows another example:

```
recursive function FindSame(Aindex,Last,Used) &
&  result(FindSameResult)
type(card) Last
integer Aindex, i
logical matched, used(5)
if( Aindex > 5 ) then
  FindSameResult = .true.
  return
endif
. . .
```

See Also

- Q to R
- FUNCTION
- ENTRY
- RECURSIVE
- Program Units and Procedures

RETURN

Statement: *Transfers control from a subprogram to the calling program unit.*

Syntax

```
RETURN [expr]
```

expr	Is a scalar expression that is converted to an integer value if necessary. The *expr* is only allowed in subroutines; it indicates an alternate return. (An alternate return is an obsolescentfeature in Fortran 95 and Fortran 90.)

Description

When a RETURN statement is executed in a function subprogram, control is transferred to the referencing statement in the calling program unit.

When a RETURN statement is executed in a subroutine subprogram, control is transferred to the first executable statement following the CALL statement that invoked the subroutine, or to the alternate return (if one is specified).

Example

The following shows how alternate returns can be used in a subroutine:

```
   CALL CHECK(A, B, *10, *20, C)
   ...
10 ...
20 ...
   SUBROUTINE CHECK(X, Y, *, *, C)
   ...
50   IF (X) 60, 70, 80
60   RETURN
70   RETURN 1
80   RETURN 2
   END
```

The value of X determines the return, as follows:

- If X < 0, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the calling program.
- If X = = 0, the first alternate return (RETURN 1) occurs and control is transferred to the statement identified with label 10.
- If X > 0, the second alternate return (RETURN 2) occurs and control is transferred to the statement identified with label 20.

Note that an asterisk (*) specifies the alternate return. An ampersand (&) can also specify an alternate return in a CALL statement, but not in a subroutine's dummy argument list.

The following shows another example:

```
      SUBROUTINE Loop
      CHARACTER in
 10   READ (*, '(A)') in
      IF (in .EQ. 'Y') RETURN
      GOTO 10
!     RETURN implied by the following statement:
      END
!The following example shows alternate returns:
      CALL AltRet (i, *10, *20, *30)
      WRITE (*, *) 'normal return'
      GOTO 40
 10   WRITE (*, *) 'I = 10'
      GOTO 40
 20   WRITE (*, *) 'I = 20'
      GOTO 40
 30   WRITE (*, *) 'I = 30'
 40   CONTINUE
      END
      SUBROUTINE AltRet (i, *, *, *)
      IF (i .EQ. 10) RETURN 1
      IF (i .EQ. 20) RETURN 2
      IF (i .EQ. 30) RETURN 3
      END
```

In this example, RETURN 1 specifies the list's first alternate-return label, which is a symbol for the actual argument *10 in the CALL statement. RETURN 2 specifies the second alternate-return label, and RETURN 3 specifies the third alternate-return label.

See Also

- Q to R

- CALL
- CASE

REWIND

Statement: *Positions a sequential or direct access file at the beginning of the file (the initial point). It takes one of the following forms:*

Syntax

```
REWIND ([UNIT=] io-unit[, ERR= label] [, IOSTAT=i-var])

REWIND io-unit
```

io-unit	(Input) Is an external unit specifier.
label	Is the label of the branch target statement that receives control if an error occurs.
i-var	(Output)Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

Description

The unit number must refer to a file on disk or magnetic tape, and the file must be open for sequential, direct, or append access.

If a REWIND is done on a direct access file, the NEXTREC specifier is assigned a value of 1.

If a file is already positioned at the initial point, a REWIND statement has no effect.

If a REWIND statement is specified for a unit that is not open, it has no effect.

Example

The following statement repositions the file connected to I/O unit 3 to the beginning of the file:

```
REWIND 3
```

Consider the following statement:

```
REWIND (UNIT=9, IOSTAT=IOS, ERR=10)
```

This statement positions the file connected to unit 9 at the beginning of the file. If an error occurs, control is transferred to the statement labeled 10, and a positive integer is stored in variable IOS.

The following shows another example:

```
WRITE (7, '(I10)') int
REWIND (7)
READ (7, '(I10)') int
```

See Also

- Q to R
- OPEN
- READ
- WRITE
- Data Transfer I/O Statements
- Branch Specifiers

REWRITE

Statement: *Rewrites the current record.*

Syntax

Formatted:

```
REWRITE (eunit, format[, iostat] [, err]) [io-list]
```

Unformatted:

```
REWRITE (eunit[, iostat][ , err]) [io-list]
```

eunit	Is an external unit specifier ([UNIT=]io-unit).
format	Is a format specifier ([FMT=]format).
iostat	Is a status specifier (IOSTAT=i-var).
err	Is a branch specifier (ERR=label) if an error condition occurs.
io-list	Is an I/O list.

Description

In the REWRITE statement, data (translated if formatted; untranslated if unformatted) is written to the current (existing) record in a file with direct access.

The current record is the last record accessed by a preceding, successful sequential or direct-access READ statement.

Between a READ and REWRITE statement, you should not specify any other I/O statement (except INQUIRE) on that logical unit. Execution of any other I/O statement on the logical unit destroys the current-record context and causes the current record to become undefined.

Only one record can be rewritten in a single REWRITE statement operation.

The output list (and format specification, if any) must not specify more characters for a record than the record size. (Record size is specified by RECLin an OPEN statement.)

If the number of characters specified by the I/O list (and format, if any) do not fill a record, blank characters are added to fill the record.

Example

In the following example, the current record (contained in the relative organization file connected to logical unit 3) is updated with the values represented by NAME, AGE, and BIRTH:

```
      REWRITE (3, 10, ERR=99) NAME, ,AGE, BIRTH
10    FORMAT (A16, I2, A8)
```

RGBTOINTEGER (W*32, W*64)

QuickWin Function: *Converts three integers specifying red, green, and blue color intensities into a four-byte RGB integer for use with RGB functions and subroutines.*

Module

USE IFQWIN

Syntax

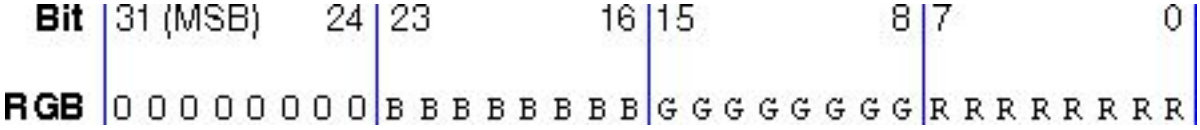
`result = RGBTOINTEGER (red, green, blue)`

red	(Input) INTEGER(4). Intensity of the red component of the RGB color value. Only the lower 8 bits of *red* are used.
green	(Input) INTEGER(4). Intensity of the green component of the RGB color value. Only the lower 8 bits of *green* are used.
blue	(Input) INTEGER(4). Intensity of the blue component of the RGB color value. Only the lower 8 bits of *blue* are used.

Results

The result type is INTEGER(4). The result is the combined RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value returned with RGBTOINTEGER, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
! Build as a QuickWin App.
USE IFQWIN
INTEGER r, g, b, rgb, result
INTEGER(2) status
r = Z'F0'
g = Z'F0'
b = 0
rgb = RGBTOINTEGER(r, g, b)
result = SETCOLORRGB(rgb)
status = ELLIPSE($GFILLINTERIOR,INT2(40), INT2(55), &
                 INT2(90), INT2(85))
END
```

See Also

- Q to R
- INTEGERTORGB
- SETCOLORRGB

- SETBKCOLORRGB
- SETPIXELRGB
- SETPIXELSRGB
- SETTEXTCOLORRGB

Building Applications: Using QuickWin Overview

RINDEX

Portability Function: *Locates the index of the last occurrence of a substring within a string.*

Module

USE IFPORT

Syntax

`result = RINDEX (string,substr)`

string	(Input) Character*(*). Original string to search.
substr	(Input) Character*(*). String to search for.

Results

The result type is INTEGER(4). The result is the starting position of the final occurrence of *substrg* in *string*. The result is zero if *substring* does not occur in *string*.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
character*80 mainstring
character*4 shortstr
integer(4) where
mainstring="Hello Hello Hello Hello There There There"
shortstr="Hello"
where=rindex(mainstring,shortstr)
! where is 19
```

See Also

- Q to R
- INDEX

RNUM

Elemental Intrinsic Function (Specific):
Converts a character string to a REAL(4) value. This function cannot be passed as an actual argument.

Syntax

```
result = RNUM (i)
```

i	(Input) Must be of type character.

Results

The result type is REAL(4). The result value is the real value represented by the character string *i*.

Example

RNUM ("821.003") has the value 821.003 of type REAL(4).

RRSPACING

Elemental Intrinsic Function (Generic):
Returns the reciprocal of the relative spacing of model numbers near the argument value.

Syntax

```
result = RRSPACING (x)
```

x	(Input) Must be of type real.

Results

The result type is the same as *x*. The result has the value $|x| \times b^{-e}| \times b^{p}$. Parameters *b*, *e*, *p* are defined in Model for Real Data.

Example

If -3.0 is a REAL(4) value, RRSPACING (-3.0) has the value 0.75×2^{24}.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = RRSPACING(3.0)  ! returns 1.258291E+07
res4 = RRSPACING(-3.0) ! returns 1.258291E+07
r2 = 487923.3
res8 = RRSPACING(r2)   ! returns 8.382458680573952E+015
END
```

See Also

- Q to R
- SPACING
- Data Representation Models

RSHIFT

Elemental Intrinsic Function (Generic): *Shifts the bits in an integer right by a specified number of positions. See ISHFT.*

RTC

Portability Function: *Returns the number of seconds elapsed since a specific Greenwich mean time.*

Module

USE IFPORT

Syntax

```
result = RTC( )
```

Results

The result type is REAL(8). The result is the number of seconds elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
real(8) s, s1, time_spent
INTEGER(4) i, j
s = RTC( )
call sleep(4)
s1 = RTC( )
time_spent = s1 - s
PRINT *, 'It took ',time_spent, 'seconds to run.'
```

See Also

- Q to R
- DATE_AND_TIME
- TIME portability routine

RUNQQ

Portability Function: *Executes another program and waits for it to complete.*

Module

USE IFPORT

Syntax

`result = RUNQQ (`*`filename`*`,`*`commandline`*`)`

filename	(Input) Character*(*). File name of a program to be executed.
commandline	(Input) Character*(*). Command-line arguments passed to the program to be executed.

Results

The result type is INTEGER(2). If the program executed with RUNQQ terminates normally, the exit code of that program is returned to the program that launched it. If the program fails, -1 is returned.

The RUNQQ function executes a new process for the operating system using the same path, environment, and resources as the process that launched it. The launching process is suspended until execution of the launched process is complete.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(2) result

result = RUNQQ('myprog', '-c -r')

END
```

See also the example in NARGS.

See Also

- Q to R
- NARGS
- SYSTEM
- NARGS

S

SAVE

Statement and Attribute: *Causes the values and definition of objects to be retained after execution of a RETURN or END statement in a subprogram.*

Syntax

The SAVE attribute can be specified in a type declaration statement or a SAVE statement, and takes one of the following forms:

Type Declaration Statement:

```
type,[att-ls,]  SAVE [, att-ls] :: object[, object] ...]
```

Statement:

```
SAVE [[::]object[, object] ...]
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
object	Is the name of an object, or the name of a common block enclosed in slashes (/*common-block-name*/).

Description

In Intel® Fortran, certain variables are given the SAVE attribute, or not, by default:

- The following variables are *not* saved by default:
  - Scalar local variables of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL without default initialization
  - Variables that are declared AUTOMATIC
  - Local variables that are allocatable arrays
  - Derived-type variables that are data initialized by default initialization of any of their components
  - RECORD variables that are data initialized by default initialization specified in its STRUCTURE declaration
- The following variables are saved by default:
  - COMMON variables
  - Scalar local variables not of intrinsic types INTEGER, REAL, COMPLEX, and LOGICAL of non-recursive subprograms
  - Non-scalar local variables of non-recursive subprograms
  - Module variables
  - Data initialized by DATA statements
- Local variables that are not described in the preceding two lists are saved by default.

NOTE. Certain compiler options, such as `-save` and `-automatic` (Linux and Mac OS X) or `/Qsave` and `/automatic` (Windows), and use of OpenMP can change the defaults.

To enhance portability and avoid possible compiler warning messages, Intel recommends that you use the SAVE statement to name variables whose values you want to preserve between subprogram invocations.

When a SAVE statement does not explicitly contain a list, all allowable items in the scoping unit are saved.

A SAVE statement cannot specify the following (their values cannot be saved):

- A blank common
- An object in a common block
- A procedure
- A dummy argument
- A function result
- An automatic object
- A PARAMETER (named) constant

Even though a common block can be included in a SAVE statement, individual variables within the common block can become undefined (or redefined) in another scoping unit.

If a common block is saved in any scoping unit of a program (other than the main program), it must be saved in every scoping unit in which the common block appears.

A SAVE statement has no effect in a main program.

Example

The following example shows a type declaration statement specifying the SAVE attribute:

```
SUBROUTINE TEST()
  REAL, SAVE :: X, Y
```

The following is an example of the SAVE statement:

```
SAVE A, /BLOCK_B/, C, /BLOCK_D/, E
```

The following shows another example:

```
  SUBROUTINE MySub
    COMMON /z/ da, in, a, idum(10)
    real(8) x,y
    ...
    SAVE x, y, /z/
! alternate declaration
    REAL(8), SAVE :: x, y
    SAVE /z/
```

See Also

- S
- COMMON
- DATA
- RECURSIVE
- MODULE
- MODULE PROCEDURE
- Type Declarations
- Compatible attributes
- SAVE value in CLOSE

SAVEIMAGE, SAVEIMAGE_W (W*32, W*64)

Graphics Functions: *Save an image from a specified portion of the screen into a Windows bitmap file.*

Module

USE IFQWIN

Syntax

`result = SAVEIMAGE (`*`filename`*`,`*`ulxcoord`*`,`*`ulycoord`*`, `*`lrxcoord`*`,`*`lrycoord`*`)`

`result = SAVEIMAGE_W (`*`filename`*`,`*`ulwxcoord`*`,`*`ulwycoord`*`, `*`lrwxcoord`*`,`*`lrwycoord`*`)`

`filename` (Input) Character*(*). Path of the bitmap file.

ulxcoord, ulycoord	(Input) INTEGER(4). Viewport coordinates for upper-left corner of the screen image to be captured.
lrxcoord, *lrycoord*	(Input) INTEGER(4). Viewport coordinates for lower-right corner of the screen image to be captured.
ulwxcoord, ulwycoord	(Input) REAL(8). Window coordinates for upper-left corner of the screen image to be captured.
lrwxcoord, lrwycoord	(Input) REAL(8). Window coordinates for lower-right corner of the screen image to be captured.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The SAVEIMAGE function captures the screen image within a rectangle defined by the upper-left and lower-right screen coordinates and stores the image as a Windows bitmap file specified by *filename*. The image is stored with a palette containing the colors displayed on the screen.

SAVEIMAGE defines the bounding rectangle in viewport coordinates. SAVEIMAGE_W defines the bounding rectangle in window coordinates.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

See Also

- S
- GETIMAGE, GETIMAGE_W
- IMAGESIZE, IMAGESIZE_W
- LOADIMAGE, LOADIMAGE_W
- PUTIMAGE, PUTIMAGE_W

Building Applications: Loading and Saving Images to Files

SCALE

Elemental Intrinsic Function (Generic):
Returns the value of the exponent part (of the model for the argument) changed by a specified value.

Syntax

```
result = SCALE (x,i)
```

x	(Input) Must be of type real.
i	(Input) Must be of type integer.

Results

The result type is the same as *x*. The result has the value *x* x b ^{i}. Parameter *b* is defined in Model for Real Data.

Example

If 3.0 is a REAL(4) value, SCALE (3.0, 2) has the value 12.0 and SCALE (3.0, 3) has the value 24.0.

The following shows another example:

```
REAL r

r = SCALE(5.2, 2)     !returns 20.8
```

See Also

- S
- LSHIFT
- Data Representation Models

SCAN

Elemental Intrinsic Function (Generic): *Scans a string for any character in a set of characters.*

Syntax

```
result = SCAN (string, set [, back] [, kind])
```

string	(Input) Must be of type character.
set	(Input) Must be of type character with the same kind parameter as *string*.
back	(Input) Must be of type logical.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is in *set*, the value of the result is the position of the leftmost character of *string* that is in *set*.

If *back* is present with the value true and *string* has at least one character that is in *set*, the value of the result is the position of the rightmost character of *string* that is in *set*.

If no character of *string* is in *set* or the length of *string* or *set* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

SCAN ('ASTRING', 'ST') has the value 2.

SCAN ('ASTRING', 'ST', BACK=.TRUE.) has the value 3.

SCAN ('ASTRING', 'CD') has the value zero.

The following shows another example:

```
INTEGER i
INTEGER array(2)
i = SCAN ('FORTRAN', 'TR')                ! returns 3
i = SCAN ('FORTRAN', 'TR', BACK = .TRUE.)  ! returns 5
i = SCAN ('FORTRAN', 'GHA')               ! returns 6
i = SCAN ('FORTRAN', 'ora')               ! returns 0
array = SCAN ((/'FORTRAN','VISUALC'/),(/'A', 'A'/))
                                          ! returns (6, 5)
! Note that when using SCAN with arrays, the string
! elements must be the same length. When using string
! constants, blank pad to make strings the same length.
! For example:
array = SCAN ((/'FORTRAN','MASM '/),(/'A', 'A'/))
                                          ! returns (6, 2)
END
```

See Also

- S
- VERIFY

SCANENV

Portability Subroutine: *Scans the environment for the value of an environment variable.*

Module

USE IFPORT

Syntax

`CALL SCANENV (envname,envtext,envvalue)`

envname	(Input) Character*(*). Contains the name of an environment variable you need to find the value for.
envtext	(Output) Character*(*). Set to the full text of the environment variable if found, or to ' ' if nothing is found.
envvalue	(Output) Character*(*). Set to the value associated with the environment variable if found or to ' ' if nothing is found. SCANENV scans for an environment variable that matches *envname* and returns the value or string it is set to.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

SCROLLTEXTWINDOW (W*32, W*64)

Graphics Subroutine: *Scrolls the contents of a text window.*

Module

USE IFQWIN

Syntax

`CALL SCROLLTEXTWINDOW (rows)`

rows	(Input) INTEGER(2). Number of rows to scroll.

The SCROLLTEXTWINDOW subroutine scrolls the text in a text window (previously defined by SETTEXTWINDOW). The default text window is the entire window.

The *rows* argument specifies the number of lines to scroll. A positive value for *rows* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling CLEARSCREEN ($GWINDOW). A value of 0 for *rows* has no effect.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as QuickWin or Standard Graphics app.
 USE IFQWIN
 INTEGER(2) row, istat
 CHARACTER(18) string
 TYPE (rccoord) oldpos
 CALL SETTEXTWINDOW (INT2(1), INT2(0), &
                     INT2(25), INT2(80))
 CALL CLEARSCREEN ( $GCLEARSCREEN )
 CALL SETTEXTPOSITION (INT2(1), INT2(1), oldpos)
 DO row = 1, 6
   string = 'Hello, World # '
   CALL SETTEXTPOSITION( row, INT2(1), oldpos )
   WRITE(string(15:16), '(I2)') row
   CALL OUTTEXT( string )
 END DO
 istat  = displaycursor($GCURSORON)
 WRITE(*,'(1x,A\)') 'Hit ENTER'
 READ (*,*) ! wait for ENTER
 !  Scroll window down 4 lines
 CALL SCROLLTEXTWINDOW(INT2( -4) )
 CALL SETTEXTPOSITION (INT2(10), INT2(18), oldpos)
 WRITE(*,'(2X,A\)') "Hit ENTER"
 READ( *,* ) ! wait for ENTER
 !  Scroll window up 5 lines
 CALL SCROLLTEXTWINDOW( INT2(5 ))
 END
```

See Also

- S
- CLEARSCREEN
- GETTEXTPOSITION
- GETTEXTWINDOW
- GRSTATUS
- OUTTEXT
- SETTEXTPOSITION
- SETTEXTWINDOW
- WRAPON

Building Applications: Displaying Character-Based Text

Building Applications: Text Coordinates

SCWRQQ

Portability Subroutine: *Returns the floating-point processor control word.*

Module

USE IFPORT

Syntax

`CALL SCWRQQ (control)`

control (Output) INTEGER(2). Floating-point processor control word.

SCRWQQ performs the same function as the run-time subroutine GETCONTROLFPQQ, and is provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

See the example in LCWRQQ.

See Also

- S
- GETCONTROLFPQQ

- LCWRQQ

SECNDS Intrinsic Procedure

Elemental Intrinsic Function (Generic):
Provides the system time of day, or elapsed time, as a floating-point value in seconds. SECNDS can be used as an intrinsic function or as a portability function. It is an intrinsic procedure unless you specify USE IFPORT.

Syntax

This function must not be passed as an actual argument. It is not a pure function, so it cannot be referenced inside a FORALL construct.

```
result = SECNDS (x)
```

x (Input) Must be of type real.

Results

The result type is the same as *x*. The result value is the time in seconds since midnight - *x*. (The function also produces correct results for time intervals that span midnight.)

The value of SECNDS is accurate to 0.01 second, which is the resolution of the system clock.

The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS LIB

Example

The following shows how to use SECNDS to perform elapsed-time computations:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)
C     CODE TO BE TIMED
      ...
      DELTA = SECNDS(T1)      ! DELTA gives the elapsed time
```

See Also

- S
- DATE_AND_TIME
- RTC
- SYSTEM_CLOCK
- TIME intrinsic procedure
- SECNDS portability routine

SECNDS Portability Routine

Portability Function: *Returns the number of seconds that have elapsed since midnight, less the value of its argument. SECNDS can be used as an intrinsic function or as a portability function. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

`result = SECNDS (time)`

time	(Input) REAL(4). Number of seconds, precise to a hundredth of a second (0.01), to be subtracted.

Results

The result type is REAL(4). The result value is the number of seconds that have elapsed since midnight, minus *time*, with a precision of a hundredth of a second (0.01).

To start the timing clock, call SECNDS with 0.0, and save the result in a local variable. To get the elapsed time since the last call to SECNDS, pass the local variable to SECNDS on the next call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
REAL(4) s
INTEGER(4) i, j
s = SECNDS(0.0)
DO I = 1, 100000
   J = J + 1
END DO
s = SECNDS(s)
PRINT *, 'It took ',s, 'seconds to run.'
```

See Also

- S
- DATE_AND_TIME
- RTC
- SYSTEM_CLOCK
- TIME portability routine
- SECNDS intrinsic procedure

SECTIONS

OpenMP* Fortran Compiler Directive: *Specifies one or more blocks of code that must be divided among threads in the team. Each section is executed once by a thread in the team.*

Syntax

```
c$OMP SECTIONS [clause[[,] clause] ... ]
[c$OMP SECTION]
   block
[c$OMP SECTION
   block]...
c$OMP END SECTIONS[NOWAIT]
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Is one of the following: • FIRSTPRIVATE (list) • LASTPRIVATE (list) • PRIVATE (list) • REDUCTION (operator | intrinsic : list)
block	Is a structured block (section) of statements or constructs. Any constituent section must also be a structured block. You cannot branch into or out of the block.

Each section of code is preceded by a SECTION directive, although the directive is optional for the first section. The SECTION directives must appear within the lexical extent of the SECTIONS and END SECTIONS directive pair.

The last section ends at the END SECTIONS directive. Threads that complete execution of their SECTIONs encounter an implied barrier at the END SECTIONS directive unless NOWAIT is specified.

SECTIONS directives must be encountered by all threads in a team or by none at all.

Example

In the following example, subroutines XAXIS, YAXIS, and ZAXIS can be executed concurrently:

```
c$OMP PARALLEL
c$OMP SECTIONS
c$OMP SECTION
      CALL XAXIS
c$OMP SECTION
      CALL YAXIS
c$OMP SECTION
      CALL ZAXIS
c$OMP END SECTIONS
c$OMP END PARALLEL
```

See Also

- S
- OpenMP Fortran Compiler Directives

SEED

Portability Subroutine: *Changes the starting point of the pseudorandom number generator.*

Module

USE IFPORT

Syntax

`CALL SEED (iseed)`

iseed	(Input) INTEGER(4). Starting point for RANDOM.

SEED uses *iseed* to establish the starting point of the pseudorandom number generator. A given seed always produces the same sequence of values from RANDOM.

If SEED is not called before the first call to RANDOM, RANDOM always begins with a seed value of one. If a program must have a different pseudorandom sequence each time it runs, pass the constant RND$TIMESEED (defined in `IFPORT.F90`) to the SEED routine before the first call to RANDOM.

This routine is not thread-safe.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
REAL myrand
CALL SEED(7531)
CALL RANDOM(myrand)
```

See Also

- S
- RANDOM

- RANDOM_SEED
- RANDOM_NUMBER

SELECT CASE...END SELECT

Statement: *Transfers program control to a selected block of statements according to the value of a controlling expression. CASE.*

Example

```
CHARACTER*1 cmdchar

. . .

Files: SELECT CASE (cmdchar)

  CASE ('0')

    WRITE (*, *) "Must retrieve one to nine files"

  CASE ('1':'9')

    CALL RetrieveNumFiles (cmdchar)

  CASE ('A', 'a')

    CALL AddEntry

  CASE ('D', 'd')

    CALL DeleteEntry

  CASE ('H', 'h')

    CALL Help

  CASE DEFAULT

    WRITE (*, *) "Command not recognized; please re-enter"

END SELECT Files
```

SELECTED_CHAR_KIND

Transformational Intrinsic Function (Generic):
Returns the value of the kind type parameter of the character set named by the argument.

Syntax

```
result = SELECTED_CHAR_KIND(name)
```

name (Input) Must be scalar and of type default character. Its value must be 'DEFAULT' or 'ASCII'.

Results

The result is a scalar of type default integer.

The result is a scalar of type default integer. The result value is 1 if NAME has the value 'DEFAULT' or 'ASCII'; otherwise, the result value is -1.

SELECTED_INT_KIND

Transformational Intrinsic Function (Generic):
Returns the value of the kind parameter of an integer data type.

Syntax

```
result = SELECTED_INT_KIND (r)
```

r (Input) Must be scalar and of type integer.

Results

The result is a scalar of type default integer. The result has a value equal to the value of the kind parameter of the integer data type that represents all values *n* in the range of values *n* with $-10^{r} < n < 10^{r}$.

If no such kind type parameter is available on the processor, the result is -1. If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range. For more information, see Model for Integer Data.

Example

SELECTED_INT_KIND (6) = 4

The following shows another example:

```
i = SELECTED_INT_KIND(8)  ! returns 4
i = SELECTED_INT_KIND(3)  ! returns 2
i = SELECTED_INT_KIND(10) ! returns 8
i = SELECTED_INT_KIND(20) ! returns -1 because 10**20
                          !   is bigger than 2**63
```

See Also

- S
- SELECTED_REAL_KIND

SELECTED_REAL_KIND

Transformational Intrinsic Function (Generic): *Returns the value of the kind parameter of a real data type.*

Syntax

```
result = SELECTED_REAL_KIND ([p] [,r])
```

p	(Input; optional) Must be scalar and of type integer.
r	(Input; optional) Must be scalar and of type integer.

At least one argument must be present.

Results

If *p* or *r* is absent, the result is as if the argument was present with the value zero.

The result is a scalar of type default integer. If both arguments are absent, the result is zero. Otherwise,the result has a value equal to a value of the kind parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least *p* digits and a decimal exponent range, as returned by the function RANGE, of at least *r*.

If no such kind type parameter is available on the processor, the result is as follows:

```
-1 if the precision is not available but the range is available
-2 if the exponent range is not available but the precision is available
-3 if neither is available
```

-4 if real types for the precision and the range are available separately but not together

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision. Intel Fortran currently does not return -4 for any combination of *p* and *r*. For more information, see Model for Real Data.

Example

SELECTED_REAL_KIND (6, 70) = 8

The following shows another example:

```
i = SELECTED_REAL_KIND(r=200)  ! returns 8
i = SELECTED_REAL_KIND(13)     ! returns 8
i = SELECTED_REAL_KIND (100, 200)  ! returns -1
i = SELECTED_REAL_KIND (13, 5000)  ! returns -2
i = SELECTED_REAL_KIND (100, 5000) ! returns -3
```

The following example gives a compile-time error:

```
i = SELECTED_REAL_KIND ()
```

The following example returns a 0 at run-time to indicate that there was an error:

```
PROGRAM TEST
CALL F ()
CONTAINS
  SUBROUTINE F (P, R)
  INTEGER, OPTIONAL :: P, R
  PRINT *, SELECTED_REAL_KIND (P=P, R=R)  ! prints 0
  END SUBROUTINE F
END PROGRAM TEST
```

See Also

- S
- SELECTED_INT_KIND

SEQUENCE

Statement: *Preserves the storage order of a derived-type definition.*

Syntax

```
SEQUENCE
```

Description

The SEQUENCE statement allows derived types to be used in common blocks and to be equivalenced.

The SEQUENCE statement appears only as part of derived-type definitions. It causes the components of the derived type to be stored in the same sequence they are listed in the type definition. If you do not specify SEQUENCE, the physical storage order is not necessarily the same as the order of components in the type definition.

If a derived type is a sequence derived type, then any other derived type that includes it must also be a sequence type.

Example

```
!DEC$ PACK:1
TYPE NUM1_SEQ
  SEQUENCE
  INTEGER(2)::int_val
  REAL(4)::real_val
  LOGICAL(2)::log_val
END TYPE NUM1_SEQ
TYPE num2_seq
   SEQUENCE
   logical(2)::log_val
   integer(2)::int_val
   real(4)::real_val
end type num2_seq
type (num1_seq) num1
type (num2_seq) num2
character*8 t, t1
equivalence (num1,t)
equivalence (num2,t1)
num1%int_val=2
num1%real_val=3.5
num1%log_val=.TRUE.
t1(1:2)=t(7:8)
t1(3:4)=t(1:2)
t1(5:8)=t(3:6)
print *, num2%int_val, num2%real_val, num2%log_val
end
```

See Also

- S
- Derived Data Types
- Data Types, Constants, and Variables

SETACTIVEQQ (W*32, W*64)

QuickWin Function: *Makes a child window active, but does not give it focus.*

Module

USE IFQWIN

Syntax

`result = SETACTIVEQQ (unit)`

unit	(Input) INTEGER(4). Unit number of the child window to be made active.

Results

The result type is INTEGER(4). The result is 1 if successful; otherwise, 0.

When a window is made active, it receives graphics output (from ARC, LINETO and OUTGTEXT, for example) but is not brought to the foreground and does not have the focus. If a window needs to be brought to the foreground, it must be given the focus. A window is given focus with FOCUSQQ, by clicking it with the mouse, or by performing I/O other than graphics on it, unless the window was opened with IOFOCUS='.FALSE.'. By default, IOFOCUS='.TRUE.', except for child windows opened as unit '*'.

The window that has the focus is always on top, and all other windows have their title bars grayed out. A window can have the focus and yet not be active and not have graphics output directed to it. Graphical output is independent of focus.

If IOFOCUS='.TRUE.', the child window receives focus prior to each READ, WRITE, PRINT, or OUTTEXT. Calls to graphics functions (such as OUTGTEXT and ARC) do not cause the focus to shift.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- S
- GETACTIVEQQ
- FOCUSQQ
- INQFOCUSQQ

Building Applications: Using QuickWin Overview

Building Applications: Giving a Window Focus and Setting the Active Window

SETBKCOLOR (W*32, W*64)

Graphics Function: *Sets the current background color index for both text and graphics.*

Module

USE IFQWIN

Syntax

```
result = SETBKCOLOR (color)
```

color (Input) INTEGER(4). Color index to set the background color to.

Results

The result type is INTEGER(4). The result is the previous background color index.

SETBKCOLOR changes the background color index for both text and graphics. The color index of text over the background color is set with SETTEXTCOLOR. The color index of graphics over the background color (used by drawing functions such as FLOODFILL and ELLIPSE) is set with SETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETBKCOLORRGB, SETCOLORRGB, and SETTEXTCOLORRGB.

Changing the background color index does not change the screen immediately. The change becomes effective when CLEARSCREEN is executed or when doing text input or output, such as with READ, WRITE, or OUTTEXT. The graphics output function OUTGTEXT does not affect the color of the background.

Generally, INTEGER(4) color arguments refer to color values and INTEGER(2) color arguments refer to color indexes. The two exceptions are GETBKCOLOR and SETBKCOLOR. The default background color index is 0, which is associated with black unless the user remaps the palette with REMAPPALETTERGB.

NOTE. The SETBKCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetBkColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetBkColor. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(4) i
i = SETBKCOLOR(14)
```

See Also

- S
- SETBKCOLORRGB
- GETBKCOLOR
- REMAPALLPALETTERGB, REMAPPALETTERGB
- SETCOLOR
- SETTEXTCOLOR

Building Applications: Setting Figure Properties

Building Applications: Using Text Colors

Building Applications: Color Mixing

SETBKCOLORRGB (W*32, W*64)

Graphics Function: *Sets the current background color to the given Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

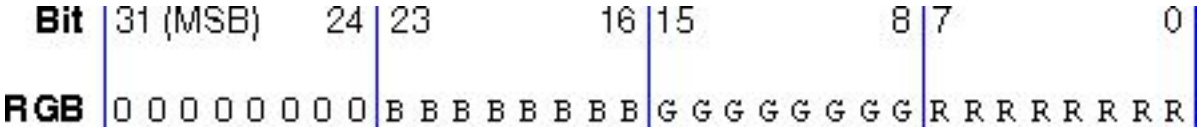
`result = SETBKCOLORRGB (color)`

color (Input) INTEGER(4). RGB color value to set the background color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous background RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETBKCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

The default background color is value 0, which is black. Changing the background color value does not change the screen immediately, but becomes effective when CLEARSCREEN is executed or when doing text input or output such as READ, WRITE, or OUTTEXT. The graphics output function OUTGTEXT does not affect the color of the background.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. The RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT) is set with SETTEXTCOLORRGB. The RGB color value of graphics over the background color (used by graphics functions such as ARC, OUTGTEXT, and FLOODFILLRGB) is set with SETCOLORRGB.

SETBKCOLORRGB (and the other RGB color selection functions SETCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(4) oldcolor
INTEGER(2) status, x1, y1, x2, y2
x1 = 80; y1 = 50
x2 = 240; y2 = 150
oldcolor = SETBKCOLORRGB(Z'FF0000') !blue
oldcolor = SETCOLORRGB(Z'FF') ! red
CALL CLEARSCREEN ($GCLEARSCREEN)
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

See Also

- S
- GETBKCOLORRGB
- SETCOLORRGB
- SETTEXTCOLORRGB
- SETPIXELRGB
- SETPIXELSRGB
- SETBKCOLOR

Building Applications: Setting Figure Properties

Building Applications: Using Text Colors

Building Applications: Color Mixing

SETCLIPRGN (W*32, W*64)

Graphics Subroutine: *Limits graphics output to part of the screen.*

Module

USE IFQWIN

Syntax

CALL SETCLIPRGN (*x1*,*y1*,*x2*,*y2*)

x1, y1	(Input) INTEGER(2). Physical coordinates for upper-left corner of clipping region.
x2, y2	(Input) INTEGER(2). Physical coordinates for lower-right corner of clipping region.

The SETCLIPRGN function limits the display of subsequent graphics output and font text output to that which fits within a designated area of the screen (the "clipping region"). The physical coordinates (x1, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangle that defines the clipping region. The SETCLIPRGN function does not change the viewport-coordinate system; it merely masks graphics output to the screen.

SETCLIPRGN affects graphics and font text output only, such as OUTGTEXT. To mask the screen for text output using OUTTEXT, use SETTEXTWINDOW.

Compatibility

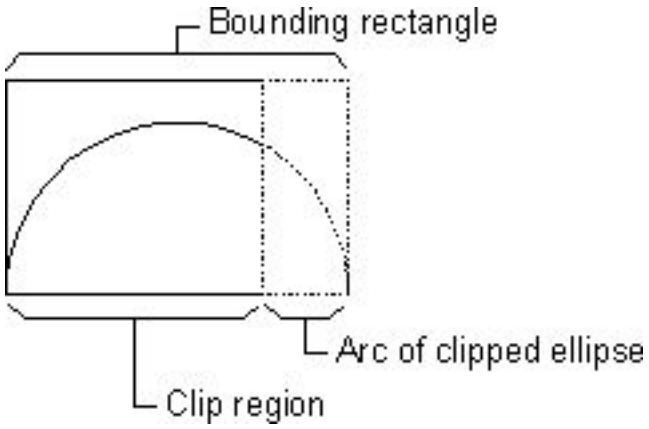
STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

This program draws an ellipse lying partly within a clipping region, as shown below.

```
! Build as QuickWin or Standard Graphics ap.
USE IFQWIN
INTEGER(2) status, x1, y1, x2, y2
INTEGER(4) oldcolor
x1 = 10;  y1 = 50
x2 = 170; y2 = 150
! Draw full ellipse in white
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
oldcolor = SETCOLORRGB(Z'FF0000') !blue
WRITE(*,*) "Hit enter"
READ(*,*)
CALL CLEARSCREEN($GCLEARSCREEN) ! clear screen
CALL SETCLIPRGN( INT2(0), INT2(0), &
                 INT2(150), INT2(125))
! only part of ellipse inside clip region drawn now
status = ELLIPSE($GBORDER, x1, y1, x2, y2)
END
```

The following shows the output of this program.



See Also

- S
- GETPHYSCOORD
- GRSTATUS
- SETTEXTWINDOW
- SETVIEWORG
- SETVIEWPORT
- SETWINDOW

Building Applications: Graphics Coordinates

Building Applications: Setting Graphics Coordinates

SETCOLOR (W*32, W*64)

Graphics Function: *Sets the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = SETCOLOR (color)
```

color	(Input) INTEGER(2). Color index to set the current graphics color to.

Results

The result type is INTEGER(2). The result is the previous color index if successful; otherwise, -1.

The SETCOLOR function sets the current graphics color index, which is used by graphics functions such as ELLIPSE. The background color index is set with SETBKCOLOR. The color index of text over the background color is set with SETTEXTCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. For access to all system colors, use SETCOLORRGB, SETBKCOLORRGB, and SETTEXTCOLORRGB.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) color, oldcolor
LOGICAL status
TYPE (windowconfig) wc
status = GETWINDOWCONFIG(wc)
color = wc%numcolors - 1
oldcolor = SETCOLOR(color)
END
```

See Also

- S
- SETCOLORRGB
- GETCOLOR
- REMAPPALETTERGB
- SETBKCOLOR
- SETTEXTCOLOR
- SETPIXEL
- SETPIXELS

Building Applications: Color Mixing

Building Applications: Setting Figure Properties

Building Applications: Using Color

Building Applications: VGA Color Palette

SETCOLORRGB (W*32, W*64)

Graphics Function: *Sets the current graphics color to the specified Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

```
result = SETCOLORRGB (color)
```

color	(Input) INTEGER(4). RGB color value to set the current graphics color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:

Bit	31 (MSB) 24	23 16	15 8	7 0
RGB	0 0 0 0 0 0 0 0	B B B B B B B B	G G G G G G G G	R R R R R R R R

Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

SETCOLORRGB sets the RGB color value of graphics over the background color, used by the following graphics functions: ARC, ELLIPSE, FLOODFILL, LINETO, OUTGTEXT, PIE, POLYGON, RECTANGLE, and SETPIXEL. SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETTEXTCOLORRGB sets the RGB color value of text over the background color (used by text functions such as OUTTEXT, WRITE, and PRINT).

SETCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB, and SETTEXTCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETCOLOR, SETBKCOLOR, and SETTEXTCOLOR) use color indexes

rather than true color values. If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a QuickWin or Standard Graphics App.
USE IFQWIN
INTEGER(2) numfonts
INTEGER(4) oldcolor
TYPE (xycoord) xy
numfonts = INITIALIZEFONTS( )
oldcolor = SETCOLORRGB(Z'0000FF')   ! red
oldcolor = SETBKCOLORRGB(Z'00FF00') ! green
CALL MOVETO(INT2(200), INT2(100), xy)
CALL OUTGTEXT("hello, world")
END
```

See Also

- S
- SETBKCOLORRGB
- SETTEXTCOLORRGB
- GETCOLORRGB
- ARC
- ELLIPSE
- FLOODFILLRGB
- SETCOLOR
- LINETO
- OUTGTEXT
- PIE

- POLYGON
- RECTANGLE
- REMAPPALETTERGB
- SETPIXELRGB
- SETPIXELSRGB

Building Applications: Color Mixing

Building Applications: Setting Figure Properties

Building Applications: Using Color

SETCONTROLFPQQ

Portability Subroutine: *Sets the value of the floating-point processor control word.*

Module

USE IFPORT

Syntax

`CALL SETCONTROLFPQQ (controlword)`

controlword (Input) INTEGER(2). Floating-point processor control word.

The floating-point control word specifies how various exception conditions are handled by the floating-point math coprocessor, sets the floating-point precision, and specifies the floating-point rounding mechanism used.

The control word can be any of the following constants (defined in `IFPORT.F90`):

Parameter name	Hex value	Description
FPCW$MCW_IC	Z'1000'	**Infinity control mask**
FPCW$AFFINE	Z'1000'	Affine infinity
FPCW$PROJECTIVE	Z'0000'	Projective infinity
FPCW$MCW_PC	Z'0300'	**Precision control mask**
FPCW$64	Z'0300'	64-bit precision
FPCW$53	Z'0200'	53-bit precision

Parameter name	Hex value	Description
FPCW$24	Z'0000'	24-bit precision
FPCW$MCW_RC	Z'0C00'	**Rounding control mask**
FPCW$CHOP	Z'0C00'	Truncate
FPCW$UP	Z'0800'	Round up
FPCW$DOWN	Z'0400'	Round down
FPCW$NEAR	Z'0000'	Round to nearest
FPCW$MCW_EM	Z'003F'	**Exception mask**
FPCW$INVALID	Z'0001'	Allow invalid numbers
FPCW$DENORMAL	Z'0002'	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	Z'0004'	Allow divide by zero
FPCW$OVERFLOW	Z'0008'	Allow overflow
FPCW$UNDERFLOW	Z'0010'	Allow underflow
FPCW$INEXACT	Z'0020'	Allow inexact precision

An exception is disabled if its control bit is set to 1. An exception is enabled if its control bit is cleared to 0.

Setting the floating-point precision and rounding mechanism can be useful if you are reusing old code that is sensitive to the floating-point precision standard used and you want to get the same results as on the old machine.

You can use GETCONTROLFPQQ to retrieve the current control word and SETCONTROLFPQQ to change the control word. Most users do not need to change the default settings. If you need to change the control word, always use SETCONTROLFPQQ to make sure that special routines handling floating-point stack exceptions and abnormal propagation work correctly.

For a full discussion of the floating-point control word, exceptions, and error handling, see *Building Applications*: The Floating-Point Environment Overview.

NOTE. The Intel® Fortran exception handler allows for software masking of invalid operations, but does not allow the math chip to mask them. If you choose to use the software masking, be aware that this can affect program performance if you compile code written for Intel Fortran with another compiler.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(2) status, control, controlo
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!     Save old control word
controlo = control
!     Clear all flags
control = control .AND. Z'0000'
!     Set new control to round up
control = control .OR. FPCW$UP
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

See Also

- S
- GETCONTROLFPQQ
- GETSTATUSFPQQ
- LCWRQQ
- SCWRQQ

- CLEARSTATUSFPQQ

Building Applications: Exception Parameters

Building Applications: Floating-Point Control Word Overview

SETDAT

Portability Function: *Sets the system date. This function is only available on Windows* and Linux* systems.*

Module

USE IFPORT

Syntax

`result = SETDAT (iyr, imon, iday)`

iyr	(Input) INTEGER(2) or INTEGER(4). Year (*xxxx*AD).
imon	(Input) INTEGER(2) or INTEGER(4). Month (1-12).
iday	(Input) INTEGER(2) or INTEGER(4). Day of the month (1-31).

Results

The result type is LOGICAL(4). The result is .TRUE. if the system date is changed; .FALSE. if no change is made.

Actual arguments of the function SETDAT can be any valid INTEGER(2) or INTEGER(4) expression.

Refer to your operating system documentation for the range of permitted dates.

NOTE. On Linux systems, you must have root privileges to execute this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
LOGICAL(4) success
success = SETDAT(INT2(1997+1), INT2(2*3), INT2(30))
END
```

See Also

- S
- GETDAT
- GETTIM
- SETTIM

SETENVQQ

Portability Function: *Sets the value of an existing environment variable, or adds and sets a new environment variable.*

Module

USE IFPORT

Syntax

`result = SETENVQQ (varname=value)`

varname=value	(Input) Character*(*). String containing both the name and the value of the variable to be added or modified. Must be in the form: *varname = value*, where *varname* is the name of an environment variable and *value* is the value being assigned to it.

Results

The result is of type LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

Environment variables define the environment in which a program executes. For example, the LIB environment variable defines the default search path for libraries to be linked with a program.

SETENVQQ deletes any terminating blanks in the string. Although the equal sign (=) is an illegal character within an environment value, you can use it to terminate *value* so that trailing blanks are preserved. For example, the string `PATH= `=sets *value* to `''`.

You can use SETENVQQ to remove an existing variable by giving a variable name followed by an equal sign with no value. For example, LIB= removes the variable LIB from the list of environment variables. If you specify a value for a variable that already exists, its value is changed. If the variable does not exist, it is created.

SETENVQQ affects only the environment that is local to the current process. You cannot use it to modify the command-level environment. When the current process terminates, the environment reverts to the level of the parent process. In most cases, this is the operating system level. However, you can pass the environment modified by SETENVQQ to any child process created by RUNQQ. These child processes get new variables and/or values added by SETENVQQ.

SETENVQQ uses the C runtime routine _putenv and GETENVQQ uses the C runtime routine getenv. From the C documentation:

getenv and _putenv use the copy of the environment pointed to by the global variable _environ to access the environment. getenv operates only on the data structures accessible to the run-time library and not on the environment segment created for the process by the operating system.

SETENVQQ and GETENVQQ will not work properly with the Windows* APIs SetEnvironmentVariable and GetEnvironmentVariable.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

LOGICAL(4) success

success = SETENVQQ("PATH=c:\mydir\tmp")

success = &

SETENVQQ("LIB=c:\mylib\bessel.lib;c:\math\difq.lib")

END
```

See Also

- S
- GETENVQQ
- RUNQQ

SETERRORMODEQQ

Portability Subroutine: *Sets the prompt mode for critical errors that by default generate system prompts.*

Module

USE IFPORT

Syntax

`CALL SETERRORMODEQQ (`*`pmode`*`)`

pmode	(Input) LOGICAL(4). Flag that determines whether a prompt is displayed when a critical error occurs.

Certain I/O errors cause the system to display an error prompt. For example, attempting to write to a disk drive with the drive door open generates an "Abort, Retry, Ignore" message. When the system starts up, system error prompting is enabled by default (*pmode*= .TRUE.). You can also enable system error prompts by calling SETERRORMODEQQ with *pmode* set to ERR$HARDPROMPT (defined in `IFPORT.F90`).

If prompt mode is turned off, critical errors that normally cause a system prompt are silent. Errors in I/O statements such as OPEN, READ, and WRITE fail immediately instead of being interrupted with prompts. This gives you more direct control over what happens when an error occurs. For example, you can use the ERR= specifier to designate an executable statement to branch to for error handling. You can also take a different action than that requested by the system prompt, such as opening a temporary file, giving a more informative error message, or exiting.

You can turn off prompt mode by setting *pmode* to .FALSE. or to the constant ERR$HARDFAIL (defined in `IFPORT.F90`).

Note that SETERRORMODEQQ affects only errors that generate a system prompt. It does not affect other I/O errors, such as writing to a nonexistent file or attempting to open a nonexistent file with STATUS='OLD'.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!PROGRAM 1
!  DRIVE B door open
OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Generates a system prompt error here and waits for the user
!  to respond to the prompt before continuing
100   WRITE(*,*) ' Continuing'
END
! PROGRAM 2
!  DRIVE B door open
 USE IFPORT
 CALL SETERRORMODEQQ(.FALSE.)
 OPEN (10, FILE = 'B:\NOFILE.DAT', ERR = 100)
!  Causes the statement at label 100 to execute
!  without system prompt
100   WRITE(*,*) ' Drive B: not available, opening      &
                  &alternative drive.'
 OPEN (10, FILE = 'C:\NOFILE.DAT')
 END
```

SETEXITQQ

QuickWin Function: *Sets a QuickWin application's exit behavior.*

Module

USE IFQWIN

Syntax

`result = SETEXITQQ (exitmode)`

exitmode (Input) INTEGER(4). Determines the program exit behavior. The following exit parameters are defined in `IFQWIN.F90`:

- QWIN$EXITPROMPT - Displays the following message box:

  `"Program exited with exit status X. Exit Window?"`

  where X is the exit status from the program.

  If Yes is entered, the application closes the window and terminates. If No is entered, the dialog box disappears and you can manipulate the windows as usual. You must then close the window manually.
- QWIN$EXITNOPERSIST - Terminates the application without displaying a message box.
- QWIN$EXITPERSIST - Leaves the application open without displaying a message box.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a negative value.

The default for both QuickWin and Standard Graphics applications is QWIN$EXITPROMPT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as QuickWin Ap
USE IFQWIN
INTEGER(4) exmode, result
WRITE(*,'(1X,A,/)') 'Please enter the exit mode 1, 2   &
                    or 3 '
READ(*,*) exmode
SELECT CASE (exmode)
  CASE (1)
    result = SETEXITQQ(QWIN$EXITPROMPT)
  CASE (2)
    result = SETEXITQQ(QWIN$EXITNOPERSIST)
  CASE (3)
    result = SETEXITQQ(QWIN$EXITPERSIST)
  CASE DEFAULT
    WRITE(*,*) 'Invalid option - checking for bad     &
                return'
    IF(SETEXITQQ( exmode ) .NE. -1) THEN
      WRITE(*,*) 'Error not returned'
    ELSE
      WRITE(*,*) 'Error code returned'
    ENDIF
END SELECT
END
```

See Also

- S
- GETEXITQQ

Building Applications: Using QuickWin Overview

SET_EXPONENT

Elemental Intrinsic Function (Generic): *Returns the value of the exponent part (of the model for the argument) set to a specified value.*

Syntax

```
result = SET_EXPONENT (x,i)
```

x	(Input) Must be of type real.
i	(Input) Must be of type integer.

Results

The result type is the same as *x*. The result has the value *x* x b $^{i\text{-}e}$. Parameters *b* and *e* are defined in Model for Real Data. If *x* has the value zero, the result is zero.

Example

If 3.0 is a REAL(4) value, SET_EXPONENT (3.0, 1) has the value 1.5.

See Also

- S
- EXPONENT
- Data Representation Models

SETFILEACCESSQQ

Portability Function: *Sets the file access mode for a specified file.*

Module

USE IFPORT

Syntax

```
result = SETFILEACCESSQQ (filename,access)
```

filename	(Input) Character*(*). Name of a file to set access for.

access (Input) INTEGER(4). Constant that sets the access. Can be any combination of the following flags, combined by an inclusive OR (such asIOR or OR):

- FILE$ARCHIVE - Marked as having been copied to a backup device.
- FILE$HIDDEN - Hidden. The file does not appear in the directory list that you can request from the command console.
- FILE$NORMAL - No special attributes (default).
- FILE$READONLY - Write-protected. You can read the file, but you cannot make changes to it.
- FILE$SYSTEM - Used by the operating system.

The flags are defined in module `IFPORT.F90`.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

To set the access value for a file, add the constants representing the appropriate access.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) permit
LOGICAL(4) result
permit = 0   ! clear permit
permit = IOR(FILE$READONLY, FILE$HIDDEN)
result = SETFILEACCESSQQ ('formula.f90', permit)
END
```

See Also

- S
- GETFILEINFOQQ

SETFILETIMEQQ

Portability Function: *Sets the modification time for a specified file.*

Module

USE IFPORT

Syntax

`result = SETFILETIMEQQ (filename,timedate)`

filename	(Input) Character*(*). Name of a file.
timedate	(Input) INTEGER(4). Time and date information, as packed by PACKTIMEQQ.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The modification time is the time the file was last modified and is useful for keeping track of different versions of the file. The process that calls SETFILETIMEQQ must have write access to the file; otherwise, the time cannot be changed. If you set *timedate* to FILE$CURTIME (defined in IFPORT.F90), SETFILETIMEQQ sets the modification time to the current system time.

If the function fails, call GETLASTERRORQQ to determine the reason. It can be one of the following:

- ERR$ACCES - Permission denied. The file's (or directory's) permission setting does not allow the specified access.
- ERR$INVAL - Invalid argument; the *timedate* argument is invalid.
- ERR$MFILE - Too many open files (the file must be opened to change its modification time).
- ERR$NOENT - File or path not found.
- ERR$NOMEM - Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(2) day, month, year
INTEGER(2) hour, minute, second, hund
INTEGER(4) timedate
LOGICAL(4) result
CALL GETDAT(year, month, day)
CALL GETTIM(hour, minute, second, hund)
CALL PACKTIMEQQ (timedate, year, month, day,    &
                 hour, minute, second)
result = SETFILETIMEQQ('myfile.dat', timedate)
END
```

See Also

- S
- PACKTIMEQQ
- UNPACKTIMEQQ
- GETLASTERRORQQ

SETFILLMASK (W*32, W*64)

Graphics Subroutine: *Sets the current fill mask to a new pattern.*

Module

USE IFQWIN

Syntax

CALL SETFILLMASK (*mask*)

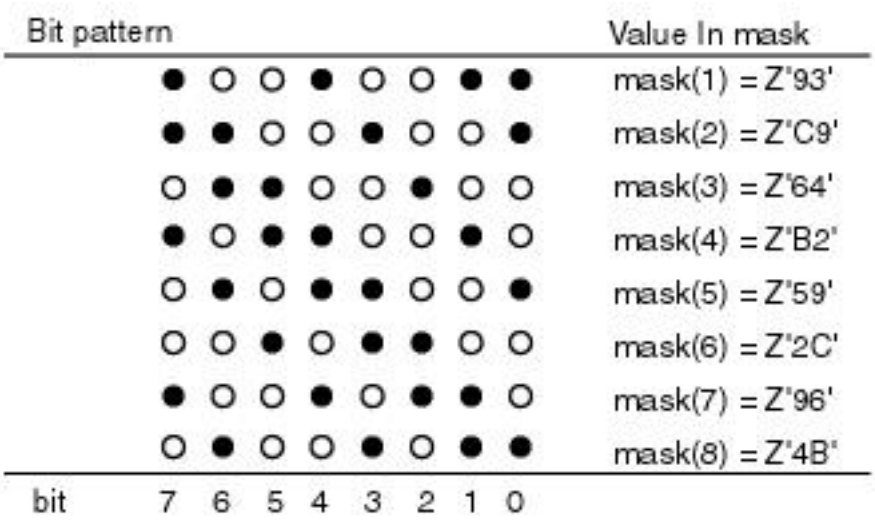
mask (Input) INTEGER(1). One-dimensional array of length 8.

There are 8 bytes in *mask*, and each of the 8 bits in each byte represents a pixel, creating an 8x8 pattern. The first element (byte) of *mask* becomes the top 8 bits of the pattern, and the eighth element (byte) of *mask* becomes the bottom 8 bits.

During a fill operation, pixels with a bit value of 1 are set to the current graphics color, while pixels with a bit value of zero are set to the current background color. The current graphics color is set with SETCOLORRGB or SETCOLOR. The 8-byte mask is replicated over the entire fill area. If no fill mask is set (with SETFILLMASK), or if the mask is all ones, solid current color is used in fill operations.

The fill mask controls the fill pattern for graphics routines (FLOODFILLRGB, PIE, ELLIPSE, POLYGON, and RECTANGLE).

To change the current fill mask, determine the array of bytes that corresponds to the desired bit pattern and set the pattern with SETFILLMASK, as in the following example.



Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

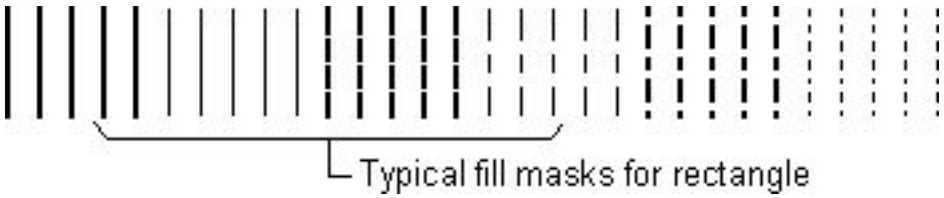
Example

This program draws six rectangles, each with a different fill mask, as shown below.

```
! Build as QuickWin or Standard Graphics Ap.
USE IFQWIN
INTEGER(1), TARGET :: style1(8) &
            /Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18',Z'18'/
INTEGER(1), TARGET :: style2(8) &
            /Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08',Z'08'/
INTEGER(1), TARGET :: style3(8) &
            /Z'18',Z'00',Z'18',Z'18',Z'18',Z'00',Z'18',Z'18'/
INTEGER(1), TARGET :: style4(8) &
            /Z'00',Z'08',Z'00',Z'08',Z'08',Z'08',Z'08',Z'08'/
INTEGER(1), TARGET :: style5(8) &
            /Z'18',Z'18',Z'00',Z'18',Z'18',Z'00',Z'18',Z'18'/
INTEGER(1), TARGET :: style6(8) &
            /Z'08',Z'00',Z'08',Z'00',Z'08',Z'00',Z'08',Z'00'/
INTEGER(1) oldstyle(8) ! Placeholder for old style
INTEGER loop
INTEGER(1), POINTER :: ptr(:)
CALL GETFILLMASK( oldstyle )
! Make 6 rectangles, each with a different fill
DO loop = 1, 6
  SELECT CASE (loop)
    CASE (1)
      ptr => style1
    CASE (2)
      ptr => style2
    CASE (3)
```

```
      ptr => style3
    CASE (4)
      ptr => style4
    CASE (5)
      ptr => style5
    CASE (6)
      ptr => style6
  END SELECT
  CALL SETFILLMASK( ptr)
  status = RECTANGLE($GFILLINTERIOR,INT2(loop*40+5), &
              INT2(90),INT2((loop+1)*40), INT2(110))
END DO
CALL SETFILLMASK( oldstyle )  ! Restore old style
READ (*,*)                    ! Wait for ENTER to be
                              ! pressed
END
```

The following shows the output of this program.



See Also

- S
- ELLIPSE
- FLOODFILLRGB
- GETFILLMASK
- PIE
- POLYGON

- RECTANGLE

Building Applications: Adding Shapes

Building Applications: Setting Figure Properties

SETFONT (W*32, W*64)

Graphics Function: *Finds a single font that matches a specified set of characteristics and makes it the current font used by the OUTGTEXT function.*

Module

USE IFQWIN

Syntax

`result = SETFONT (options)`

options	(Input) Character*(*). String describing font characteristics (see below for details).

Results

The result type is INTEGER(2). The result is the index number (*x* as used in the `nx` option) of the font if successful; otherwise, -1.

The SETFONT function searches the list of available fonts for a font matching the characteristics specified in *options* . If a font matching the characteristics is found, it becomes the current font. The current font is used in all subsequent calls to the OUTGTEXT function. There can be only one current font.

The *options* argument consists of letter codes, as follows, that describe the desired font. The argument is neither case sensitive nor position sensitive.

`t'` *fontname*`'`	Name of the desired typeface. It can be any installed font.
`h`*y*	Character height, where *y* is the number of pixels.
`w`*x*	Select character width, where *x* is the number of pixels.

f	Select only a fixed-space font (do not use with the p characteristic).
p	Select only a proportional-space font (do not use with the f characteristic).
v	Select only a vector-mapped font (do not use with the r characteristic). Roman, Modern, and Script are examples of vector-mapped fonts, also called plotter fonts. True Type fonts (for example, Arial, Symbol, and Times New Roman) are not vector-mapped.
r	Select only a raster-mapped (bitmapped) font (do not use with the v characteristic). Courier, Helvetica, and Palatino are examples of raster-mapped fonts, also called screen fonts. True Type fonts are not raster-mapped.
e	Select the bold text format. This parameter is ignored if the font does not allow the bold format.
u	Select the underline text format. This parameter is ignored if the font does not allow underlining.
i	Select the italic text format. This parameter is ignored if the font does not allow italics.
b	Select the font that best fits the other parameters specified.
n*x*	Select font number *x*, where *x* is less than or equal to the value returned by the INTIALIZEFONTS function.

You can specify as many options as you want, except with n*x*, which should be used alone. If you specify options that are mutually exclusive (such as the pairs f/p or r/v), the SETFONT function ignores them. There is no error detection for incompatible parameters used with n*x*.

If the b option is specified and at least one font is initialized, SETFONT sets a font and returns 0 to indicate success.

In selecting a font, the SETFONT routine uses the following criteria, rated from highest precedence to lowest:

1. Pixel height
2. Typeface
3. Pixel width
4. Fixed or proportional font

You can also specify a pixel width and height for fonts. If you choose a nonexistent value for either and specify the b option, SETFONT chooses the closest match.

A smaller font size has precedence over a larger size. If you request Arial 12 with best fit, and only Arial 10 and Arial 14 are available, SETFONT selects Arial 10.

If you choose a nonexistent value for pixel height and width, the SETFONT function applies a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if you specify the roption (raster-mapped font), or if you request a specific typeface and do not specify the b option (best-fit).

If you specify the n*x* parameter, SETFONT ignores any other specified options and supplies only the font number corresponding to *x*.

If a height is given, but not a width, SETFONT computes the a width to preserve the correct font proportions.

If a width is given, but not a height, SETFONT uses a default height, which may vary from font type to font type. This may lead to characters that appear distorted, particularly when a very wide width is specified. This behavior is the same as that of the Windows* API CreateFontIndirect. A sample program is provided below showing you how to calculate the correct height for a given width.

The font functions affect only OUTGTEXT and the current graphics position; no other Fortran Graphics Library output functions are affected by font usage.

For each window you open, you must call INITIALIZEFONTS before calling SETFONT. INITIALIZEFONTS needs to be executed after each new child window is opened in order for a subsequent SETFONT call to be successful.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
!  Build as a Graphics ap.
USE IFQWIN
INTEGER(2) fontnum, numfonts
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
!  Set typeface to Arial, character height to 18,
!  character width to 10, and italic
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Demo text')
END
```

Another example follows:

```
! The following program shows you how to compute
!  an appropriate font height for a given font width
!
!  Build as a Graphics ap.
 USE IFQWIN
 INTEGER(2) fontnum, numfonts
 TYPE (xycoord) pos
 TYPE (rccoord) rcc
 TYPE (FONTINFO) info
 CHARACTER*11 str, str1
 CHARACTER*22 str2
 real rh
 integer h, inw
 str = "t'Arial'bih"
 str1= " "
 numfonts = INITIALIZEFONTS ( )
 ! Default both height and width.  This seems to work
 ! properly.  From this setting get the ratio between
 ! height and width.
 fontnum = SETFONT ("t'Arial'")
 ireturn = GETFONTINFO(info)
 rh =  real(info%pixheight)/real(info%avgwidth)
 ! Now calculate the height for a width of 40
 write(*,*) 'Input desired width:'
 read(*,*) inw
 h =int(inw*rh)
 write(str1,'(I3.3)') h
 str2 = str//str1
```

```
  print *,str2
  fontnum = SETFONT (str2)
  CALL MOVETO (INT2(10), INT2(50), pos)
  CALL OUTGTEXT('ABCDEFGabcdefg12345!@#$%')
  CALL MOVETO (INT2(10), INT2(50+10+h), pos)
  CALL OUTGTEXT('12345678901234567890123 4')
  ireturn = GETFONTINFO(info)
  call settextposition(4,1, rcc)
  print *,  info%avgwidth, info%pixheight
 END
```

See Also

- S
- GETFONTINFO
- GETGTEXTEXTENT
- GRSTATUS
- OUTGTEXT
- INITIALIZEFONTS
- SETGTEXTROTATION

Building Applications: Setting the Font and Displaying Text

Building Applications: SHOWFONT.F90 Example

SETGTEXTROTATION (W*32, W*64)

Graphics Subroutine: *Sets the orientation angle of the font text output in degrees. The current orientation is used in calls to OUTGTEXT.*

Module

USE IFQWIN

Syntax

CALL SETGTEXTROTATION (*degree-tenths*)

degree-tenths (Input) INTEGER(4). Angle of orientation, in tenths of degrees, of the font text output.

The orientation of the font text output is set in tenths of degrees. Horizontal is 0°, and angles increase counterclockwise so that 900 (90°) is straight up, 1800 (180°) is upside down and left, 2700 (270°) is straight down, and so forth. If the user specifies a value greater than 3600 (360°), the subroutine takes a value equal to:

```
MODULO (user-specified tenths of degrees, 3600)
```

Although SETGTEXTROTATION accepts arguments in tenths of degrees, only increments of one full degree differ visually from each other on the screen.

Bitmap fonts cannot be rotated; TruType fonts should be used instead.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) fontnum, numfonts
INTEGER(4) oldcolor, deg
TYPE (xycoord) pos
numfonts = INITIALIZEFONTS ( )
fontnum = SETFONT ('t''Arial''h18w10i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT('Straight text')
deg = -1370
CALL SETGTEXTROTATION(deg)
oldcolor = SETCOLORRGB(Z'008080')
CALL OUTGTEXT('Slanted text')
END
```

See Also

- S

- GETGTEXTROTATION

Building Applications: Selecting Display Options

SETLINESTYLE (W*32, W*64)

Graphics Subroutine: *Sets the current line style to a new line style.*

Module

USE IFQWIN

Syntax

```
CALL SETLINESTYLE (mask)
```

mask	(Input) INTEGER(2). Desired Quickwin line-style mask. (See the table below.)

The mask is mapped to the style that most closely equivalences the percentage of the bits in the mask that are set. The style produces lines that cover a certain percentage of the pixels in that line.

SETLINESTYLE sets the style used in drawing a line. You can choose from the following styles:

QuickWin Mask	Internal Windows* Style	Selection Criteria	Appearance
0xFFFF	PS_SOLID	16 bits on	_____
0xEEEE	PS_DASH	11 to 15 bits on	----------
0xECEC	PS_DASHDOT	10 bits on	-.-.-.-.-.-.-.-.-.-.
0xECCC	PS_DASHDOTDOT	9 bits on	-..-..-..-..-..-..-..
0xAAAA	PS_DOT	1 to 8 bits on	.....
0x0000	PS_NULL	0 bits on	

SETLINESTYLE affects the drawing of straight lines as in LINETO, POLYGON, and RECTANGLE, but not the drawing of curved lines as in ARC, ELLIPSE, or PIE.

The current graphics color is set with SETCOLORRGB or SETCOLOR. SETWRITEMODE affects how the line is displayed.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
 USE IFQWIN
 INTEGER(2)     status, style
 TYPE (xycoord) xy
 style = Z'FFFF'
 CALL SETLINESTYLE(style)
 CALL MOVETO(INT2(50), INT2(50), xy )
 status = LINETO(INT2(300), INT2(300))
 END
```

See Also

- S
- GETLINESTYLE
- GRSTATUS
- LINETO
- POLYGON
- RECTANGLE
- SETCOLOR
- SETWRITEMODE

Building Applications: Adding Shapes

Building Applications: Drawing Lines on the Screen

Building Applications: Setting Figure Properties

SETMESSAGEQQ (W*32, W*64)

QuickWin Subroutine: *Changes QuickWin status messages, state messages, and dialog box messages.*

Module

USE IFQWIN

Syntax

```
CALL SETMESSAGEQQ (msg,id)
```

msg (Input) Character*(*). Message to be displayed. Must be a regular Fortran string, not a C string. Can include multibyte characters.

id (Input) INTEGER(4). Identifier of the message to be changed. The following table shows the messages that can be changed and their identifiers:

Id	Message
QWIN$MSG_TERM	"Program terminated with exit code"
QWIN$MSG_EXITQ	"\nExit Window?"
QWIN$MSG_FINISHED	"Finished"
QWIN$MSG_PAUSED	"Paused"
QWIN$MSG_RUNNING	"Running"
QWIN$MSG_FILEOPENDLG	"Text Files(*.txt), *.txt; Data Files(*.dat), *.dat; All Files(*.*), *.*;"
QWIN$MSG_BMPSAVEDLG	"Bitmap Files(*.bmp), *.bmp; All Files(*.*), *.*;"
QWIN$MSG_INPUTPEND	"Input pending in"
QWIN$MSG_PASTEINPUTPEND	"Paste input pending"

Id	Message
QWIN$MSG_MOUSEINPUTPEND	"Mouse input pending in"
QWIN$MSG_SELECTTEXT	"Select Text in"
QWIN$MSG_SELECTGRAPHICS	"Select Graphics in"
QWIN$MSG_PRINTABORT	"Error! Printing Aborted."
QWIN$MSG_PRINTLOAD	"Error loading printer driver"
QWIN$MSG_PRINTNODEFAULT	"No Default Printer."
QWIN$MSG_PRINTDRIVER	"No Printer Driver."
QWIN$MSG_PRINTINGERROR	"Print: Printing Error."
QWIN$MSG_PRINTING	"Printing"
QWIN$MSG_PRINTCANCEL	"Cancel"
QWIN$MSG_PRINTINPROGRESS	"Printing in progress..."
QWIN$MSG_HELPNOTAVAIL	"Help Not Available for Menu Item"
QWIN$MSG_TITLETEXT	"Graphic"

QWIN$MSG_FILEOPENDLG and QWIN$MSG_BMPSAVEDLG control the text in file choosing dialog boxes and have the following syntax:

"file description, file designation"

You can change any string produced by QuickWin by calling SETMESSAGEQQ with the appropriate *id*. This includes status messages displayed at the bottom of a QuickWin application, state messages (such as "Paused"), and dialog box messages. These messages can include multibyte characters. (For more information on multibyte characters, see *Building Applications*: Using National Language Support Routines Overview.) To change menu messages, use MODIFYMENUSTRINGQQ.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
print*, "Hello"
CALL SETMESSAGEQQ('Changed exit text', QWIN$MSG_EXITQ)
```

See Also

- S
- MODIFYMENUSTRINGQQ

Building Applications: Changing Status Bar and State Messages

Building Applications: Displaying Character-Based Text

SETMOUSECURSOR (W*32, W*64)

Quickwin Function: *Sets the shape of the mouse cursor for the window in focus.*

Module

USE IFQWIN

USE IFWIN

Syntax

`oldcursor = SETMOUSECURSOR (`*`newcursor`*`)`

newcursor	(Input) INTEGER(4). A Windows HCURSOR value. For many predefined shapes, LoadCursor(0, shape) is a convenient way to get a legitimate value. See the list of predefined shapes below. A value of zero prevents the cursor from being displayed.

Results

The result type is INTEGER(4). This is also an HCURSOR Value. The result is the previous cursor value.

The window in focus at the time SETMOUSECURSOR is called has its cursor changed to the specified value. Once changed, the cursor retains its shape until another call to SETMOUSECURSOR.

In Standard Graphics applications, units 5 and 6 (the default screen input and output units) are always considered to be in focus.

The following predefined values for cursor shapes are available:

Predefined Value	Cursor Shape
IDC_APPSTARTING	Standard arrow and small hourglass
IDC_ARROW	Standard arrow
IDC_CROSS	Crosshair
IDC_IBEAM	Text I-beam
IDC_ICON	Obsolete value
IDC_NO	Slashed circle
IDC_SIZE	Obsolete value; use IDC_SIZEALL
IDC_SIZEALL	Four-pointed arrow
IDC_SIZENESW	Double-pointed arrow pointing northeast and southwest
IDC_SIZENS	Double-pointed arrow pointing north and south
IDC_SIZENWSE	Double-pointed arrow pointing northwest and southeast
IDC_SIZEWE	Double-pointed arrow pointing west and east
IDC_UPARROW	Vertical arrow
IDC_WAIT	Hour glass

A LoadCursor must be done on these values before they can be used by SETMOUSECURSOR.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as Standard Graphics or QuickWin
    use ifqwin
    use ifwin
    integer*4  cursor, oldcursor
    write(6,*) 'The cursor will now be changed to an hour glass shape'
    write(6,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_WAIT)
    oldcursor = SetMouseCursor(cursor)
    read(5,*)
    write(6,*) 'The cursor will now be changed to a cross-hair shape'
    write(6,*) 'Hit <return> to see the next change'
    cursor = LoadCursor(0, IDC_CROSS)
    oldcursor = SetMouseCursor(cursor)
    read(5,*)
    write(6,*) 'The cursor will now be turned off'
    write(6,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(0)
    read(5,*)
    write(6,*) 'The cursor will now be turned on'
    write(6,*) 'Hit <return> to see the next change'
    oldcursor = SetMouseCursor(oldcursor)
    read(5,*)
  stop
  end
```

SETPIXEL, SETPIXEL_W (W*32, W*64)

Graphics Functions: *Set a pixel at a specified location to the current graphics color index.*

Module

USE IFQWIN

Syntax

```
result = SETPIXEL (x,y)
result = SETPIXEL_W (wx, wy)
```

x, y	(Input) INTEGER(2). Viewport coordinates for target pixel.
wx, wy	(Input) REAL(8). Window coordinates for target pixel.

Results

The result type is INTEGER(2). The result is the previous color index of the target pixel if successful; otherwise, -1 (for example, if the pixel lies outside the clipping region).

SETPIXEL sets the specified pixel to the current graphics color index. The current graphics color index is set with SETCOLOR and retrieved with GETCOLOR. The non-RGB color functions (such as SETCOLOR and SETPIXELS) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit Red-Green-Blue (RGB) value with an RGB color function, rather than a palette index with a non-RGB color function. SETPIXELRGB and SETPIXELRGB_W give access to the full color capacity of the system by using direct color values rather than indexes to a palette.

NOTE. The SETPIXEL routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetPixel routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetPixel. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) status, x, y
status = SETCOLOR(INT2(2))
x = 10
! Draw pixels.
DO y = 50, 389, 3
   status = SETPIXEL( x, y )
   x = x + 2
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

- S
- SETPIXELRGB
- GETPIXEL
- SETPIXELS
- GETPIXELS
- GETCOLOR
- SETCOLOR

Building Applications: Using Color

SETPIXELRGB, SETPIXELRGB_W (W*32, W*64)

Graphics Functions: *Set a pixel at a specified location to the specified Red-Green-Blue (RGB) color value.*

Module

USE IFQWIN

Syntax

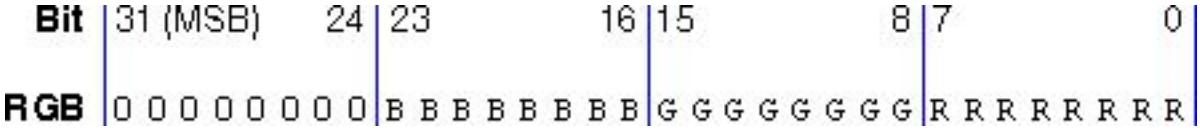
```
result = SETPIXELRGB (x,y,color)
result = SETPIXELRGB_W (x,y,color)
```

x, y	(Input) INTEGER(2). Viewport coordinates for target pixel.
wx, wy	(Input) REAL(8). Window coordinates for target pixel.
color	(Input) INTEGER(4). RGB color value to set the pixel to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous RGB color value of the pixel.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETPIXELRGB or SETPIXELRGB_W, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

If any of the pixels are outside the clipping region, those pixels are ignored.

SETPIXELRGB (and the other RGB color selection functions such as SETPIXELSRGB, SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) x, y
INTEGER(4) color
DO i = 10, 30, 10
  SELECT CASE (i)
    CASE(10)
      color = Z'0000FF'
    CASE(20)
      color = Z'00FF00'
    CASE (30)
      color = Z'FF0000'
  END SELECT
! Draw pixels.
  DO y = 50, 180, 2
    status = SETPIXELRGB( x, y, color )
    x      = x + 2
  END DO
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

- S
- GETPIXELRGB
- GETPIXELSRGB
- SETCOLORRGB
- SETPIXELSRGB

Building Applications: Color Mixing

Building Applications: Drawing a Sine Curve

Building Applications: Using Color

SETPIXELS (W*32, W*64)

Graphics Subroutine: *Sets the color indexes of multiple pixels.*

Module

USE IFQWIN

Syntax

`CALL SETPIXELS (n, x, y, color)`

n	(Input) INTEGER(4). Number of pixels to set. Sets the number of elements in the other arguments.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of pixels to set.
color	(Input) INTEGER(2). Array containing color indexes to set the pixels to.

SETPIXELS sets the pixels specified in the arrays *x* and *y* to the color indexes in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELS with n less than 1 are also ignored. SETPIXELS is a much faster way to set multiple pixel color indexes than individual calls to SETPIXEL.

Unlike SETPIXELS, SETPIXELSRGB gives access to the full color capacity of the system by using direct color values rather than indexes to a palette. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
    USE IFQWIN
    INTEGER(2) color(9)
    INTEGER(2) x(9), y(9), i
    DO i = 1, 9
      x(i) = 20 * i
      y(i) = 10 * i
      color(i) = INT2(i)
    END DO
    CALL SETPIXELS(9, x, y, color)
    END
```

See Also

- S
- GETPIXELS
- SETPIXEL
- SETPIXELSRGB

SETPIXELSRGB (W*32, W*64)

Graphics Subroutine: *Sets multiple pixels to the given Red-Green-Blue (RGB) color.*

Module

USE IFQWIN

Syntax

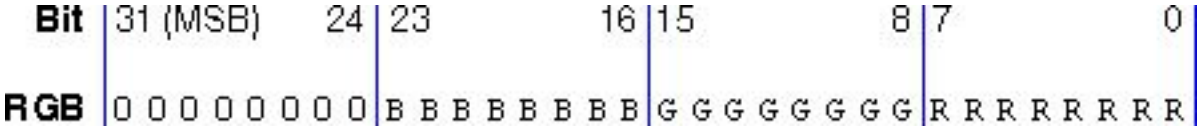
CALL SETPIXELSRGB (*n*,*x*,*y*,*color*)

n	(Input) INTEGER(4). Number of pixels to be changed. Determines the number of elements in arrays *x* and *y*.
x, y	(Input) INTEGER(2). Parallel arrays containing viewport coordinates of the pixels to set.

color	(Input) INTEGER(4). Array containing the RGB color values to set the pixels to. Range and result depend on the system's display adapter.

SETPIXELSRGB sets the pixels specified in the arrays *x* and *y* to the RGB color values in *color*. These arrays are parallel: the first element in each of the three arrays refers to a single pixel, the second element refers to the next pixel, and so on.

In each RGB color value, each of the three color values, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you set with SETPIXELSRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

A good use for SETPIXELSRGB is as a buffering form of SETPIXELRGB, which can improve performance substantially. The example code shows how to do this.

If any of the pixels are outside the clipping region, those pixels are ignored. Calls to SETPIXELSRGB with *n* less than 1 are also ignored.

SETPIXELSRGB (and the other RGB color selection functions such as SETPIXELRGB and SETCOLORRGB) sets colors to values chosen from the entire available range. The non-RGB color functions (such as SETPIXELS and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Buffering replacement for SetPixelRGB and
! SetPixelRGB_W. This can improve performance by
! doing batches of pixels together.
USE IFQWIN
PARAMETER (I$SIZE = 200)
INTEGER(4) bn, bc(I$SIZE), status
INTEGER(2) bx(I$SIZE),by(I$SIZE)
bn = 0
DO i = 1, I$SIZE
  bn = bn + 1
  bx(bn) = i
  by(bn) = i
  bc(bn) = GETCOLORRGB()
  status = SETCOLORRGB(bc(bn)+1)
END DO
CALL SETPIXELSRGB(bn,bx,by,bc)
END
```

See Also

- S
- GETPIXELSRGB
- SETPIXELRGB
- GETPIXELRGB
- SETPIXELS

Building Applications: Color Mixing

SETTEXTCOLOR (W*32, W*64)

Graphics Function: *Sets the current text color index.*

Module

USE IFQWIN

Syntax

```
result = SETTEXTCOLOR (index)
```

index (Input) INTEGER(2). Color index to set the text color to.

Results

The result type is INTEGER(2). The result is the previous text color index.

SETTEXTCOLOR sets the current text color index. The default value is 15, which is associated with white unless the user remaps the palette. GETTEXTCOLOR returns the text color index set by SETTEXTCOLOR. SETTEXTCOLOR affects text output with OUTTEXT, WRITE, and PRINT.

The background color index is set with SETBKCOLOR and returned with GETBKCOLOR. The color index of graphics over the background color is set with SETCOLOR and returned with GETCOLOR. These non-RGB color functions use color indexes, not true color values, and limit the user to colors in the palette, at most 256. To access all system colors, use SETTEXTCOLORRGB, SETBKCOLORRGB, and SETCOLORRGB.

NOTE. The SETTEXTCOLOR routine described here is a QuickWin routine. If you are trying to use the Microsoft* Platform SDK version of the SetTextColor routine by including the IFWIN module, you need to specify the routine name as MSFWIN$SetTextColor. For more information, see *Building Applications*: Special Naming Convention for Certain QuickWin and Win32 Graphics Routines.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) oldtc
oldtc = SETTEXTCOLOR(INT2(2)) ! green
WRITE(*,*) "hello, world"
END
```

See Also

- S
- GETTEXTCOLOR
- REMAPPALETTERGB
- SETCOLOR
- SETTEXTCOLORRGB

Building Applications: Color Mixing

Building Applications: Using Color

Building Applications: Using Text Colors

SETTEXTCOLORRGB (W*32, W*64)

Graphics Function: *Sets the current text color to the specified Red-Green-Blue (RGB) value.*

Module

USE IFQWIN

Syntax

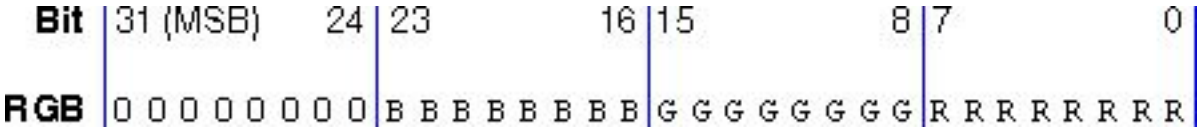
`result = SETTEXTCOLORRGB (color)`

color	(Input) INTEGER(4). RGB color value to set the text color to. Range and result depend on the system's display adapter.

Results

The result type is INTEGER(4). The result is the previous text RGB color value.

In each RGB color value, each of the three colors, red, green, and blue, is represented by an eight-bit value (2 hex digits). In the value you specify with SETTEXTCOLORRGB, red is the rightmost byte, followed by green and blue. The RGB value's internal structure is as follows:



Larger numbers correspond to stronger color intensity with binary 1111111 (hex Z'FF') the maximum for each of the three components. For example, Z'0000FF' yields full-intensity red, Z'00FF00' full-intensity green, Z'FF0000' full-intensity blue, and Z'FFFFFF' full-intensity for all three, resulting in bright white.

SETTEXTCOLORRGB sets the current text RGB color. The default value is Z'00FFFFFF', which is full-intensity white. SETTEXTCOLORRGB sets the color used by OUTTEXT, WRITE, and PRINT. It does not affect the color of text output with the OUTGTEXT font routine. Use SETCOLORRGB to change the color of font output.

SETBKCOLORRGB sets the RGB color value of the current background for both text and graphics. SETCOLORRGB sets the RGB color value of graphics over the background color, used by the graphics functions such as ARC, FLOODFILLRGB, and OUTGTEXT.

SETTEXTCOLORRGB (and the other RGB color selection functions SETBKCOLORRGB and SETCOLORRGB) sets the color to a value chosen from the entire available range. The non-RGB color functions (SETTEXTCOLOR, SETBKCOLOR, and SETCOLOR) use color indexes rather than true color values.

If you use color indexes, you are restricted to the colors available in the palette, at most 256. Some display adapters (SVGA and true color) are capable of creating 262,144 (256K) colors or more. To access any available color, you need to specify an explicit RGB value with an RGB color function, rather than a palette index with a non-RGB color function.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(4) oldtc
oldtc = SETTEXTCOLORRGB(Z'000000FF')
WRITE(*,*) 'I am red'
oldtc = SETTEXTCOLORRGB(Z'0000FF00')
CALL OUTTEXT ('I am green'//CHAR(13)//CHAR(10))
oldtc = SETTEXTCOLORRGB(Z'00FF0000')
PRINT *, 'I am blue'
END
```

See Also

- S
- SETBKCOLORRGB
- SETCOLORRGB
- GETTEXTCOLORRGB
- GETWINDOWCONFIG
- OUTTEXT

Building Applications: Color Mixing

Building Applications: Using Color

Building Applications: Using Text Colors

SETTEXTCURSOR (W*32, W*64)

Graphics Function: *Sets the height and width of the text cursor (the caret) for the window in focus.*

Module

USE IFQWIN

Syntax

`result = SETTEXTCURSOR (`*`newcursor`*`)`

newcursor (Input) INTEGER(2). The leftmost 8 bits specify the width of the cursor, and the rightmost 8 bits specify the height of the cursor. These dimensions can range from 1 to 8, and represent a fraction of the current character cell size. For example:

- Z'0808' - Specifies the full character cell; this is the default size.
- Z'0108' - Specifies 1/8th of the character cell width, and 8/8th (or all) of the character cell height.

If either of these dimensions is outside the range 1 to 8, it is forced to 8.

Results

The result type is INTEGER(2); it is the previous text cursor value in the same format as *newcursor*.

NOTE. After calling SETTEXTCURSOR, you must call DISPLAYCURSOR($GCURSORON) to actually see the cursor.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
use IFQWIN
integer(2) oldcur
integer(2) istat
type(rccoord) rc
open(10,file='user')
istat  = displaycursor($GCURSORON)
write(10,*) 'Text cursor is now character cell size, the default.'
read(10,*)
write(10,*) 'Setting text cursor to wide and low.'
oldcur = settextcursor(Z'0801')
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to high and narrow.'
oldcur = settextcursor(Z'0108')
istat  = displaycursor($GCURSORON)
read(10,*)
write(10,*) 'Setting text cursor to a dot.'
oldcur = settextcursor(Z'0101')
istat  = displaycursor($GCURSORON)
read(10,*)
end
```

See Also

- S
- DISPLAYCURSOR

SETTEXTPOSITION (W*32, W*64)

Graphics Subroutine: *Sets the current text position to a specified position relative to the current text window.*

Module

USE IFQWIN

Syntax

CALL SETTEXTPOSITION (*row*,*column*,*t*)

row	(Input) INTEGER(2). New text row position.
column	(Input) INTEGER(2). New text column position.
t	(Output) Derived type `rccoord`. Previous text position. The derived type `rccoord`is defined in `IFQWIN.F90` as follows:

```
TYPE rccoord
  INTEGER(2) row ! Row coordinate
  INTEGER(2) col ! Column coordinate
END TYPE rccoord
```

Subsequent text output with the OUTTEXT function (as well as standard console I/O statements, such as PRINT and WRITE) begins at the point (*row*, *column*).

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
TYPE (rccoord) curpos
WRITE(*,*) "Original text position"
CALL SETTEXTPOSITION (INT2(6), INT2(5), curpos)
WRITE (*,*) 'New text position'
END
```

See Also

- S
- CLEARSCREEN
- GETTEXTPOSITION
- OUTTEXT
- SCROLLTEXTWINDOW
- SETTEXTWINDOW
- WRAPON

Building Applications: Displaying Character-Based Text

Building Applications: Text Coordinates

Building Applications: Using Text Colors

SETTEXTWINDOW (W*32, W*64)

Graphics Subroutine: *Sets the current text window.*

Module

USE IFQWIN

Syntax

```
CALL SETTEXTWINDOW (r1,c1,r2,c2)
```

r1, c1	(Input) INTEGER(2). Row and column coordinates for upper-left corner of the text window.
r2, c2	(Input) INTEGER(2). Row and column coordinates for lower-right corner of the text window.

SETTEXTWINDOW specifies a window in row and column coordinates where text output to the screen using OUTTEXT, WRITE, or PRINT will be displayed. You set the text location within this window with SETTEXTPOSITION.

Text is output from the top of the window down. When the window is full, successive lines overwrite the last line.

SETTEXTWINDOW does not affect the output of the graphics text routine OUTGTEXT. Use the SETVIEWPORT function to control the display area for graphics output.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
TYPE (rccoord) curpos
CALL SETTEXTWINDOW(INT2(5), INT2(1), INT2(7), &
                   INT2(40))
CALL SETTEXTPOSITION (INT2(5), INT2(5), curpos)
WRITE(*,*) "Only two lines in this text window"
WRITE(*,*) "so this line will be overwritten"
WRITE(*,*) "by this line"
END
```

See Also

- S
- GETTEXTPOSITION
- GETTEXTWINDOW
- GRSTATUS
- OUTTEXT
- SCROLLTEXTWINDOW
- SETTEXTPOSITION
- SETVIEWPORT
- WRAPON

Building Applications: Displaying Character-Based Text

SETTIM

Portability Function: *Sets the system time in your programs. This function is only available on Windows* and Linux* systems.*

Module

USE IFPORT

Syntax

```
result = SETTIM (ihr, imin, isec, i100th)
```

ihr	(Output) INTEGER(4) or INTEGER(2). Hour (0-23).
imin	(Output) INTEGER(4) or INTEGER(2). Minute (0-59).
isec	(Output) INTEGER(4) or INTEGER(2). Second (0-59).
i100th	(Output) INTEGER(4) or INTEGER(2). Hundredths of a second (0-99).

Results

The result type is LOGICAL(4). The result is .TRUE. if the system time is changed; .FALSE. if no change is made.

NOTE. On Linux systems, you must have root privileges to execute this function.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
LOGICAL(4) success
success = SETTIM(INT2(21),INT2(53+3),&
                 INT2(14*2),INT2(88))
END
```

See Also

- S
- GETDAT
- GETTIM
- SETDAT

SETVIEWORG (W*32, W*64)

Graphics Subroutine: *Moves the viewport-coordinate origin (0, 0) to the specified physical point.*

Module

USE IFQWIN

Syntax

```
CALL SETVIEWORG (x,y,t)
```

x, y	(Input) INTEGER(2). Physical coordinates of new viewport origin.

t	(Output) Derived type `xycoord`. Physical coordinates of the previous viewport origin. The derived type `xycoord`is defined in `IFQWIN.F90` as follows:
	`TYPE xycoord`
	`  INTEGER(2) xcoord ! x-coordinate`
	`  INTEGER(2) ycoord ! y-coordinate`
	`END TYPE xycoord`
	The `xycoord`type variable *t*, defined in `IFQWIN.F90`, returns the physical coordinates of the previous viewport origin.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
TYPE ( xycoord ) xy
CALL SETVIEWORG(INT2(30), INT2(30), xy)
```

See Also

- S
- GETCURRENTPOSITION
- GETPHYSCOORD

- GETVIEWCOORD
- GETWINDOWCOORD
- GRSTATUS
- SETCLIPRGN
- SETVIEWPORT

Building Applications: Drawing Lines on the Screen

Building Applications: Graphics Coordinates

Building Applications: Setting Graphics Coordinates

SETVIEWPORT (W*32, W*64)

Graphics Subroutine: *Redefines the graphics viewport by defining a clipping region in the same manner as SETCLIPRGN and then setting the viewport-coordinate origin to the upper-left corner of the region.*

Module

USE IFQWIN

Syntax

```
CALL SETVIEWPORT (x1,y1,x2,y2)
```

x1, y1	(Input) INTEGER(2). Physical coordinates for upper-left corner of viewport.
x2, y2	(Input) INTEGER(2). Physical coordinates for lower-right corner of viewport.

The physical coordinates (*x1*, *y1*) and (*x2*, *y2*) are the upper-left and lower-right corners of the rectangular clipping region. Any window transformation done with the SETWINDOW function is relative to the viewport, not the entire screen.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) upx, upy
INTEGER(2) downx, downy
upx = 0
upy = 30
downx= 250
downy = 100
CALL SETVIEWPORT(upx, upy, downx, downy)
```

See Also

- S
- GETVIEWCOORD
- GETPHYSCOORD
- GRSTATUS
- SETCLIPRGN
- SETVIEWORG
- SETWINDOW

Building Applications: Drawing Lines on the Screen

Building Applications: Graphics Coordinates

Building Applications: Setting Graphics Coordinates

SETWINDOW (W*32, W*64)

Graphics Function: *Defines a window bound by the specified coordinates.*

Module

USE IFQWIN

Syntax

```
result = SETWINDOW (finvert,wx1,wy1,wx2,wy2)
```

finvert	(Input) LOGICAL(2). Direction of increase of the y-axis. If *finvert* is .TRUE., the y-axis increases from the window bottom to the window top (as Cartesian coordinates). If *finvert* is .FALSE., the y-axis increases from the window top to the window bottom (as pixel coordinates).
wx1, wy1	(Input) REAL(8). Window coordinates for upper-left corner of window.
wx2, wy2	(Input) REAL(8). Window coordinates for lower-right corner of window.

Results

The result type is INTEGER(2). The result is nonzero if successful; otherwise, 0 (for example, if the program that calls SETWINDOW is not in a graphics mode).

The SETWINDOW function determines the coordinate system used by all window-relative graphics routines. Any graphics routines that end in _W (such as ARC_W, RECTANGLE_W, and LINETO_W) use the coordinate system set by SETWINDOW.

Any window transformation done with the SETWINDOW function is relative to the viewport, not the entire screen.

An arc drawn using inverted window coordinates is not an upside-down version of an arc drawn with the same parameters in a noninverted window. The arc is still drawn counterclockwise, but the points that define where the arc begins and ends are inverted.

If *wx1* equals *wx2* or *wy1* equals *wy2*, SETWINDOW fails.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) status
LOGICAL(2) invert /.TRUE./
REAL(8) upx /0.0/, upy /0.0/
REAL(8) downx /1000.0/, downy /1000.0/
status = SETWINDOW(invert, upx, upy, downx, downy)
```

See Also

- S
- GETWINDOWCOORD
- SETCLIPRGN
- SETVIEWORG
- SETVIEWPORT
- GRSTATUS
- ARC_W
- LINETO_W
- MOVETO_W
- PIE_W
- POLYGON_W
- RECTANGLE_W

Building Applications: Graphics Coordinates

Building Applications: Setting Graphics Coordinates

SETWINDOWCONFIG (W*32, W*64)

QuickWin Function: *Sets the properties of a child window.*

Module

USE IFQWIN

Syntax

```
result = SETWINDOWCONFIG (wc)
```

wc (Input) Derived type `windowconfig`. Contains window properties. The `windowconfig` derived type is defined in `IFQWIN.F90` as follows:

```
TYPE windowconfig
  INTEGER(2) numxpixels         ! Number of pixels on x-axis.
  INTEGER(2) numypixels         ! Number of pixels on y-axis.
  INTEGER(2) numtextcols        ! Number of text columns
available.
  INTEGER(2) numtextrows       ! Number of text rows available.
  INTEGER(2) numcolors          ! Number of color indexes.
  INTEGER(4) fontsize           ! Size of default font. Set to
                                !   QWIN$EXTENDFONT when
specifying
                                !   extended attributes, in
which
                                !   case extendfontsize sets
the
                                !   font size.
  CHARACTER(80) title               ! The window title.
  INTEGER(2) bitsperpixel      ! The number of bits per pixel.
  INTEGER(2) numvideopages          ! Unused.
  INTEGER(2) mode               ! Controls scrolling mode.
  INTEGER(2) adapter                ! Unused.
  INTEGER(2) monitor                ! Unused.
  INTEGER(2) memory                 ! Unused.
  INTEGER(2) environment            ! Unused.
! The next three parameters provide extended font
! attributes.
  CHARACTER(32) extendfontname      ! The name of the desired
font.
  INTEGER(4) extendfontsize     ! Takes the same values as
```

```
fontsize,
                                    ! when fontsize is set to
                                    ! QWIN$EXTENDFONT.
  INTEGER(4) extendfontattributes   ! Font attributes such as
bold
                                       ! and italic.
END TYPE windowconfig
```

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The following value can be used to configure a QuickWin window so that it will show the last line written and the text cursor (if it is on):

```
wc%mode = QWIN$SCROLLDOWN
```

Note that if you scroll the window to another position, you will have to scroll back to the last line to see your input.

The following values can be used with SETWINDOWCONFIG extended fonts:

Table 908: Style:

QWIN$EXTENDFONT_NORMAL	Gives no underline, no italic, and a font weight of 400 out of 1000.
QWIN$EXTENDFONT_UNDERLINE	Gives underlined characters.
QWIN$EXTENDFONT_BOLD	Gives a font weight of 700 out of 1000.
QWIN$EXTENDFONT_ITALIC	Gives italic characters.

Table 909: Pitch:

QWIN$EXTENDFONT_FIXED_PITCH	QuickWin default. Equal character widths.
QWIN$EXTENDFONT_VARIABLE_PITCH	Variable character widths.

Table 910: Font Families:

QWIN$EXTENDFONT_FF_ROMAN	Variable stroke width, serifed. Times Roman, Century Schoolbook, etc.

QWIN$EXTENDFONT_FF_SWISS	Variable stroke width, sans-serifed. Helvetica, Swiss, etc.
QWIN$EXTENDFONT_FF_MODERN	QuickWin default. Constant stroke width, serifed or sans-serifed. Pica, Elite, Courier, etc.
QWIN$EXTENDFONT_FF_SCRIPT	Cursive, etc.
QWIN$EXTENDFONT_FF_DECORATIVE	Old English, etc.

Table 911: Character Sets:

QWIN$EXTENDFONT_ANSI_CHARSET	QuickWin default.
QWIN$EXTENDFONT_OEM_CHARSET	Use this to get Microsoft* LineDraw.

Using QWIN$EXTENDFONT_OEM_CHARSET with the font name 'MS LineDraw'C will get the old DOS-style character set with symbols that can be used to draw lines and boxes. The pitch and font family items can be specified to help guide the font matching algorithms used by CreateFontIndirect, the Windows* API used by SETWINDOWCONFIG.

If you use SETWINDOWCONFIG to set the variables in `windowconfig` to -1, the function sets the highest resolution possible for your system, given the other fields you specify, if any. You can set the actual size of the window by specifying parameters that influence the window size: the number of x and y pixels, the number of rows and columns, and the font size. If you do not call SETWINDOWCONFIG, the window defaults to the best possible resolution and a font size of 8x16. The number of colors available depends on the video driver used.

If you use SETWINDOWCONFIG, you should specify a value for each field (-1 or your own value for the numeric fields and a C string for the title, for example, "words of text"C). Using SETWINDOWCONFIG with only some fields specified can result in useless values for the unspecified fields.

If you request a configuration that cannot be set, SETWINDOWCONFIG returns .FALSE. and calculates parameter values that will work and are as close as possible to the requested configuration. A second call to SETWINDOWCONFIG establishes the adjusted values; for example:

```
status = SETWINDOWCONFIG(wc)
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

If you specify values for all four of the size parameters, *numxpixels*, *numypixel*, *numtextcols*, and *numtextrows*, the font size is calculated by dividing these values.The default font is Courier New and the default font size is 8x16. There is no restriction on font size, except that the window must be large enough to hold it.

Under Standard Graphics, the application attempts to start in Full Screen mode with no window decoration (window decoration includes scroll bars, menu bar, title bar, and message bar) so that the maximum resolution can be fully used. Otherwise, the application starts in a window. You can use ALT+ENTER at any time to toggle between the two modes.

If you are in Full Screen mode and the resolution of the window does not match the resolution of the video driver, graphics output will be slow compared to drawing in a window.

NOTE. You must call DISPLAYCURSOR($GCURSORON) to make the cursor visible after calling SETWINDOWCONFIG.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
TYPE (windowconfig) wc
LOGICAL status /.FALSE./
! Set the x & y pixels to 800X600 and font size to 8x12
wc%numxpixels  = 800
wc%numypixels  = 600
wc%numtextcols = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%title= "This is a test"C
wc%fontsize = Z'0008000C'
status = SETWINDOWCONFIG(wc)  ! attempt to set configuration with above values
   ! if attempt fails, set with system estimated values
if (.NOT.status) status = SETWINDOWCONFIG(wc)
```

See Also

- S
- DISPLAYCURSOR
- GETWINDOWCONFIG

Building Applications: Accessing Window Properties

Building Applications: Creating Child Windows

Building Applications: Graphics Coordinates

Building Applications: Selecting Display Options

Building Applications: Setting Graphics Coordinates

Building Applications: Setting the Graphics Mode

Building Applications: Using Fonts from the Graphics Library Overview

Building Applications: VGA Color Palette

SETWINDOWMENUQQ (W*32, W*64)

QuickWin Function: *Sets a top-level menu as the menu to which a list of current child window names is appended.*

Module

USE IFQWIN

Syntax

```
result = SETWINDOWMENUQQ (menuID)
```

menuID	(Input) INTEGER(4). Identifies the menu to hold the child window names, starting with 1 as the leftmost menu.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The list of current child window names can appear in only one menu at a time. If the list of windows is currently in a menu, it is removed from that menu. By default, the list of child windows appears at the end of the Window menu.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
TYPE (windowconfig) wc
LOGICAL(4) result, status /.FALSE./
! Set title for child window
wc%numxpixels  = -1
wc%numypixels  = -1
wc%numtextcols = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%fontsize    = -1
wc%title= "I am child window name"C
if (.NOT.status) status = SETWINDOWCONFIG(wc)
! put child window list under menu 3 (View)
result = SETWINDOWMENUQQ(3)
END
```

See Also

- S
- APPENDMENUQQ

Building Applications: Using QuickWin Overview

Building Applications: Customizing QuickWin Applications

Building Applications: Program Control of Menus

SETWRITEMODE (W*32, W*64)

Graphics Function: *Sets the current logical write mode, which is used when drawing lines with the LINETO, POLYGON, and RECTANGLE functions.*

Module

USE IFQWIN

Syntax

```
result = SETWRITEMODE (wmode)
```

wmode (Input) INTEGER(2). Write mode to be set. One of the following symbolic constants (defined in `IFQWIN.F90`):

- $GPSET - Causes lines to be drawn in the current graphics color. (Default)
- $GAND - Causes lines to be drawn in the color that is the logical AND of the current graphics color and the current background color.
- $GOR - Causes lines to be drawn in the color that is the logical OR of the current graphics color and the current background color.
- $GPRESET - Causes lines to be drawn in the color that is the logical NOT of the current graphics color.
- $GXOR - Causes lines to be drawn in the color that is the logical exclusive OR (XOR) of the current graphics color and the current background color.

In addition, one of the following binary raster operation constants can be used (described in the online documentation for the Windows* API SetROP2):

- $GR2_BLACK
- $GR2_NOTMERGEPEN
- $GR2_MASKNOTPEN
- $GR2_NOTCOPYPEN (same as $GPRESET)
- $GR2_MASKPENNOT
- $GR2_NOT
- $GR2_XORPEN (same as $GXOR)
- $GR2_NOTMASKPEN
- $GR2_MASKPEN (same as $GAND)
- $GR2_NOTXORPEN
- $GR2_NOP

- $GR2_MERGENOTPEN
- $GR2_COPYPEN (same as $GPSET)
- $GR2_MERGEPENNOT
- $GR2_MERGEPEN (same as $GOR)
- $GR2_WHITE

Results

The result type is INTEGER(2). The result is the previous write mode if successful; otherwise, -1.

The current graphics color is set with SETCOLORRGB (or SETCOLOR) and the current background color is set with SETBKCOLORRGB (or SETBKCOLOR). As an example, suppose you set the background color to yellow (Z'00FFFF') and the graphics color to purple (Z'FF00FF') with the following commands:

```
oldcolor = SETBKCOLORRGB(Z'00FFFF')
CALL CLEARSCREEN($GCLEARSCREEN)
oldcolor = SETCOLORRGB(Z'FF00FF')
```

If you then set the write mode with the $GAND option, lines are drawn in red (Z'0000FF'); with the $GOR option, lines are drawn in white (Z'FFFFFF'); with the $GXOR option, lines are drawn in turquoise (Z'FFFF00'); and with the $GPRESET option, lines are drawn in green (Z'00FF00'). Setting the write mode to $GPSET causes lines to be drawn in the graphics color.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
! Build as a Graphics ap.
USE IFQWIN
INTEGER(2) result, oldmode
INTEGER(4) oldcolor
TYPE (xycoord) xy
oldcolor = SETBKCOLORRGB(Z'00FFFF')
CALL CLEARSCREEN ($GCLEARSCREEN)
oldcolor = SETCOLORRGB(Z'FF00FF')
CALL MOVETO(INT2(0), INT2(0), xy)
result = LINETO(INT2(200), INT2(200)) ! purple
oldmode = SETWRITEMODE( $GAND)
CALL MOVETO(INT2(50), INT2(0), xy)
result = LINETO(INT2(250), INT2(200)) ! red
END
```

See Also

- S
- GETWRITEMODE
- GRSTATUS
- LINETO
- POLYGON
- PUTIMAGE
- RECTANGLE
- SETCOLOR
- SETLINESTYLE

Building Applications: Setting Figure Properties

SETWSIZEQQ (W*32, W*64)

QuickWin Function: *Sets the size and position of a window.*

Module

USE IFQWIN

Syntax

`result = SETWSIZEQQ (unit, winfo)`

unit (Input) INTEGER(4). Specifies the window unit. Unit numbers 0, 5, and 6 refer to the default startup window only if the program does not explicitly open them with the OPEN statement. To set the size of the frame window (as opposed to a child window), set *unit* to the symbolic constant QWIN$FRAMEWINDOW (defined in IFQWIN.F90).
When called from INITIALSETTINGS, SETWSIZEQQ behaves slightly differently than when called from a user routine after initialization. See below under Results.

winfo (Input) Derived type qwinfo. Physical coordinates of the window's upper-left corner, and the current or maximum height and width of the window's client area (the area within the frame). The derived type qwinfois defined in IFQWIN.F90 as follows:

```
TYPE QWINFO
   INTEGER(2) TYPE  ! request type
   INTEGER(2) X     ! x coordinate for upper left
   INTEGER(2) Y     ! y coordinate for upper left
   INTEGER(2) H     ! window height
   INTEGER(2) W     ! window width
END TYPE QWINFO
```

This function's behavior depends on the value of QWINFO%TYPE, which can be any of the following:

- QWIN$MIN - Minimizes the window.
- QWIN$MAX - Maximizes the window.

- QWIN$RESTORE - Restores the minimized window to its previous size.
- QWIN$SET - Sets the window's position and size according to the other values in `qwinfo`.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, nonzero (unless called from INITIALSETTINGS). If called from INITIALSETTINGS, the following occurs:

- SETWSIZEQQ always returns -1.
- Only QWIN$SET will work.

The position and dimensions of child windows are expressed in units of character height and width. The position and dimensions of the frame window are expressed in screen pixels.

The height and width specified for a frame window reflects the actual size in pixels of the frame window *including* any borders, menus, and status bar at the bottom.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(4)     result
INTEGER(2)     numfonts, fontnum
TYPE (qwinfo)  winfo
TYPE (xycoord) pos
! Maximize frame window
winfo%TYPE = QWIN$MAX
result =     SETWSIZEQQ(QWIN$FRAMEWINDOW, winfo)
! Maximize child window
result =   SETWSIZEQQ(0, winfo)
numfonts = INITIALIZEFONTS( )
fontnum =  SETFONT ('t''Arial''h50w34i')
CALL MOVETO (INT2(10), INT2(30), pos)
CALL OUTGTEXT("BIG Window")
END
```

See Also

- S
- GETWSIZEQQ
- INITIALSETTINGS

Building Applications: Using QuickWin Overview

Building Applications: Controlling Size and Position of Windows

SHAPE

Inquiry Intrinsic Function (Generic): *Returns the shape of an array or scalar argument.*

Syntax

```
result = SHAPE (source [, kind])
```

source	(Input) Is a scalar or array. It may be of any data type. It must not be an assumed-size array, a disassociated pointer, or an allocatable array that is not allocated.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is a rank-one integer array whose size is equal to the rank of *source*. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The value of the result is the shape of *source*.

The setting of compiler options specifying integer size can affect this function.

Example

SHAPE (2) has the value of a rank-one array of size zero.

If B is declared as B(2:4, -3:1), then SHAPE (B) has the value (3, 5).

The following shows another example:

```
INTEGER VEC(2)
REAL array(3:10, -1:3)
VEC = SHAPE(array)
WRITE(*,*) VEC ! prints   8       5
END
!
! Check if a mask is conformal with an array
REAL, ALLOCATABLE :: A(:,:,:)
LOGICAL, ALLOCATABLE :: MASK(:,:,:)
INTEGER B(3), C(3)
LOGICAL conform
ALLOCATE (A(5, 4, 3))
ALLOCATE (MASK(3, 4, 5))
! Check if MASK and A allocated. If they are, check
! that they have the same shape (conform).
IF(ALLOCATED(A) .AND. ALLOCATED(MASK)) THEN
  B = SHAPE(A); C = SHAPE(MASK)
  IF ((B(1) .EQ. C(1)) .AND. (B(2) .EQ. C(2))       &
       .AND. (B(3) .EQ. C(3))) THEN
     conform = .TRUE.
  ELSE
     conform = .FALSE.
  END IF
END IF
WRITE(*,*) conform  ! prints F
END
```

See Also

- S
- SIZE

SHARED Clause

Parallel Directive Clause: *Specifies variables that will be shared by all the threads in a team.*

Syntax

```
SHARED (list)
```

list	Is the name of one or more variables or common blocks that are accessible to the scoping unit. Subobjects cannot be specified. Each name must be separated by a comma, and a named common block must appear between slashes (/ /).

All threads within a team access the same storage area for SHARED data.

SHIFTL

Elemental Intrinsic Function (Specific): *Logically shifts an integer left by a specified number of bits. This function cannot be passed as an actual argument.*

Syntax

```
result = SHIFTL (ivalue,ishift)
```

ivalue	(Input) Must be of type integer. This is the value to be shifted.
ishift	(Input) Must be of type integer. The value must be positive. This value is the number of positions to shift.

Results

The result type is the same as *ivalue*. The result is the value of *ivalue* shifted left by *ishift* bit positions. Bits shifted off the left end are lost; zeros are shifted in from the opposite end.

SHIFTL (i, j) is the same as ISHFT (i, j).

See Also

- S

- ISHFT

SHIFTR

Elemental Intrinsic Function (Specific): *Logically shifts an integer right by a specified number of bits. This function cannot be passed as an actual argument.*

Syntax

```
result = SHIFTR (ivalue,ishift)
```

ivalue	(Input) Must be of type integer. This is the value to be shifted.
ishift	(Input) Must be of type integer. The value must be positive. This value is the number of positions to shift.

Results

The result type is the same as *ivalue*. The result is the value of *ivalue* shifted right by *ishift* bit positions. Bits shifted off the right end are lost; zeros are shifted in from the opposite end.

SHIFTR (i, j) is the same as ISHFT (i, -j).

See Also

- S
- ISHFT

SHORT

Portability Function: *Converts an INTEGER(4) argument to INTEGER(2) type.*

Module

USE IFPORT

Syntax

```
result = SHORT (int4)
```

int4	(Input) INTEGER(4). Value to be converted.

Results

The result type is INTEGER(2). The result is equal to the lower 16 bits of *int4*. If the *int4* value is greater than 32,767, the converted INTEGER(2) value is not equal to the original.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
      USE IFPORT
      INTEGER(4) this_one
      INTEGER(2) that_one
      READ(*,*) this_one
      THAT_ONE = SHORT(THIS_ONE)
      WRITE(*,10) THIS_ONE, THAT_ONE
 10   FORMAT (X," Long integer: ", I16, " Short integer: ", I16)
      END
```

See Also

- S
- INT
- Type Declarations

SIGN

Elemental Intrinsic Function (Generic):
Returns the absolute value of the first argument times the sign of the second argument.

Syntax

`result = SIGN (a, b)`

a	(Input) Must be of type integer or real.
b	(Input) Must have the same type and kind parameters as *a*.

Results

The result type is the same as *a*. The value of the result is as follows:

- | *a* | if *b* > zero and -| *a* | if *b* < zero.
- | *a* | if *b* is of type integer and is zero.
- If *b* is of type real and zero and compiler option `assume minus0` is not specified, the value of the result is | *a* |.
- If *b* is of type real and zero and compiler option `assume minus0` is specified, the processor can distinguish between positive and negative real zero and the following occurs:
  - If *b* is positive real zero, the value of the result is | *a* |.
  - If *b* is negative real zero, the value of the result is -| *a* |.

Specific Name	Argument Type	Result Type
BSIGN	INTEGER(1)	INTEGER(1)
IISIGN[1]	INTEGER(2)	INTEGER(2)
ISIGN [2]	INTEGER(4)	INTEGER(4)
KISIGN	INTEGER(8)	INTEGER(8)
SIGN [3]	REAL(4)	REAL(4)
DSIGN [3,4]	REAL(8)	REAL(8)
QSIGN	REAL(16)	REAL(16)

[1] Or HSIGN.

[2] Or JISIGN. For compatibility with older versions of Fortran, ISIGN is treated as a generic function.

[3] The setting of compiler options specifying real size can affect SIGN and DSIGN .

[4] The setting of compiler options specifying double size can affect DSIGN.

Example

SIGN (4.0, -6.0) has the value -4.0.

SIGN (-5.0, 2.0) has the value 5.0.

The following shows another example:

```
c = SIGN (5.2, -3.1)   ! returns -5.2
c = SIGN (-5.2, -3.1)  ! returns -5.2
c = SIGN (-5.2, 3.1)   ! returns  5.2
```

See Also

- S
- ABS
- assume minus0 compiler option

SIN

Elemental Intrinsic Function (Generic):
Produces the sine of x.

Syntax

```
result = SIN (x)
```

x	(Input) Must be of type real or complex. It must be in radians and is treated as modulo 2*pi. If *x* is of type complex, its real part is regarded as a value in radians.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SIN	REAL(4)	REAL(4)
DSIN	REAL(8)	REAL(8)
QSIN	REAL(16)	REAL(16)
CSIN [1]	COMPLEX(4)	COMPLEX(4)
CDSIN[2]	COMPLEX(8)	COMPLEX(8)
CQSIN	COMPLEX(16)	COMPLEX(16)

Specific Name	Argument Type	Result Type

[1]The setting of compiler options specifying real size can affect CSIN.

[2]This function can also be specified as ZSIN.

Example

SIN (2.0) has the value 0.9092974.

SIN (0.8) has the value 0.7173561.

SIND

Elemental Intrinsic Function (Generic):
Produces the sine of x.

Syntax

```
result = SIND (x)
```

x (Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SIND	REAL(4)	REAL(4)
DSIND	REAL(8)	REAL(8)
QSIND	REAL(16)	REAL(16)

Example

SIND (2.0) has the value 3.4899496E-02.

SIND (0.8) has the value 1.3962180E-02.

SINH

Elemental Intrinsic Function (Generic):
Produces a hyperbolic sine.

Syntax

```
result = SINH (x)
```

x (Input) Must be of type real.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
SINH	REAL(4)	REAL(4)
DSINH	REAL(8)	REAL(8)
QSINH	REAL(16)	REAL(16)

Example

SINH (2.0) has the value 3.626860.

SINH (0.8) has the value 0.8881060.

SIGNAL

Portability Function: *Controls interrupt signal handling. Changes the action for a specified signal.*

Module

USE IFPORT

Syntax

```
result = SIGNAL (signum, proc, flag)
```

signum (Input) INTEGER(4). Number of the signal to change. The numbers and symbolic names are listed in a table below.

proc	(Input) Name of a signal-processing routine. It must be declared EXTERNAL. This routine is called only if *flag* is negative.
flag	(Input) INTEGER(4). If negative, the user's *proc* routine is called. If 0, the signal retains its default action; if 1, the signal should be ignored.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 and IA-64 architectures. The result is the previous value of *proc* associated with the specified signal. For example, if the previous value of *proc* was SIG_IGN, the return value is also SIG_IGN. You can use this return value in subsequent calls to SIGNAL if the signal number supplied is invalid, if the flag value is greater than 1, or to restore a previous action definition.

A return value of SIG_ERR indicates an error, in which case a call to IERRNO returns EINVAL. If the signal number supplied is invalid, or if the flag value is greater than 1, SIGNAL returns -(EINVAL) and a call to IERRNO returns EINVAL.

An initial signal handler is in place at startup for SIGFPE (signal 8); its address is returned the first time SIGNAL is called for SIGFPE. No other signals have initial signal handlers.

Be careful when you use SIGNALQQ or the C signal function to set a handler, and then use the Portability SIGNAL function to retrieve its value. If SIGNAL returns an address that was not previously set by a call to SIGNAL, you cannot use that address with either SIGNALQQ or C's signal function, nor can you call it directly. You can, however, use the return value from SIGNAL in a subsequent call to SIGNAL. This allows you to restore a signal handler, no matter how the original signal handler was set.

All signal handlers are called with a single integer argument, that of the signal number actually received. Usually, when a process receives a signal, it terminates. With the SIGNAL function, a user procedure is called instead. The signal handler routine must accept the signal number integer argument, even if it does not use it. If the routine does not accept the signal number argument, the stack will not be properly restored after the signal handler has executed.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. You cannot use the following kinds of signal-handler routines:

- Routines that perform low-level (such as FGETC) or high-level (such as READ) I/O.
- Heap routines or any routine that uses the heap routines (such as MALLOC and ALLOCATE).
- Functions that generate a system call (such as TIME).

The following table lists signals, their names and values:

Symbolic name	Number	Description
SIGABRT	6	Abnormal termination
SIGFPE	8	Floating-point error
SIGKILL[1]	9	Kill process
SIGILL	4	Illegal instruction
SIGINT	2	CTRL+C signal
SIGSEGV	11	Illegal storage access
SIGTERM	15	Termination request

[1]SIGKILL can be neither caught nor ignored.

The default action for all signals is to terminate the program with exit code.

ABORT does not assert the SIGABRT signal. The only way to assert SIGABRT or SIGTERM is to use KILL.

SIGNAL can be used to catch SIGFPE exceptions, but it cannot be used to access the error code that caused the SIGFPE. To do this, use SIGNALQQ instead.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
    USE IFPORT
    EXTERNAL h_abort
    INTEGER(4) iret1, iret2, procnum
    iret1 = SIGNAL(SIGABRT, h_abort, -1)
    WRITE(*,*) 'Set signal handler. Return = ', iret1
    iret2 = KILL(procnum, SIGABRT)
    WRITE(*,*) 'Raised signal. Return = ', iret2
    END
!
!  Signal handler routine
!
    INTEGER(4) FUNCTION h_abort (sig_num)
    INTEGER(4) sig_num
    WRITE(*,*) 'In signal handler for SIG$ABORT'
    WRITE(*,*) 'signum = ', sig_num
    h_abort = 1
    END
```

See Also

- S
- SIGNALQQ

SIGNALQQ

Portability Function: *Registers the function to be called if an interrupt signal occurs.*

Module

USE IFPORT

Syntax

```
result = SIGNALQQ (sig,func)
```

sig	(Input) INTEGER(2). Interrupt type. One of the following constants, defined in IFPORT.F90: • SIG$ABORT - Abnormal termination • SIG$FPE - Floating-point error • SIG$ILL - Illegal instruction • SIG$INT - CTRL+CSIGNAL • SIG$SEGV - Illegal storage access • SIG$TERM - Termination request
func	(Input) Function to be executed on interrupt. It must be declared EXTERNAL.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 and IA-64 architectures. The result is a positive integer if successful; otherwise, -1 (SIG$ERR).

SIGNALQQ installs the function *func* as the handler for a signal of the type specified by *sig*. If you do not install a handler, the system by default terminates the program with exit code 3 when an interrupt signal occurs.

The argument *func* is the name of a function and must be declared with either the EXTERNAL or IMPLICIT statements, or have an explicit interface. A function described in an INTERFACE block is EXTERNAL by default, and does not need to be declared EXTERNAL.

NOTE. All signal-handler functions must be declared with the cDEC$ ATTRIBUTES C option.

When an interrupt occurs, except a SIG$FPE interrupt, the *sig* argument SIG$INT is passed to *func*, and then *func* is executed.

When a SIG$FPE interrupt occurs, the function *func* is passed two arguments: SIG$FPE and the floating-point error code (for example, FPE$ZERODIVIDE or FPE$OVERFLOW) which identifies the type of floating-point exception that occurred. The floating-point error codes begin with the prefix FPE$ and are defined in IFPORT.F90. Floating-point exceptions are described and discussed in *Building Applications*: The Floating-Point Environment Overview.

If *func* returns, the calling process resumes execution immediately after the point at which it received the interrupt signal. This is true regardless of the type of interrupt or operating mode.

Because signal-handler routines are normally called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. Therefore, do not call heap routines or any routine that uses the heap routines (for example, I/O routines, ALLOCATE, and DEALLOCATE).

To test your signal handler routine you can generate interrupt signals by calling RAISEQQ, which causes your program either to branch to the signal handlers set with SIGNALQQ, or to perform the system default behavior if SIGNALQQ has set no signal handler.

The example below shows a signal handler for SIG$ABORT. A sample signal handler for SIG$FPE is given in *Building Applications*: Handling Floating-Point Exceptions.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!  This program shows a signal handler for
!  SIG$ABORT
USE IFPORT
INTERFACE
  FUNCTION h_abort (signum)
    !DEC$ ATTRIBUTES C :: h_abort
    INTEGER(4) h_abort
    INTEGER(2) signum
  END FUNCTION
END INTERFACE
INTEGER(2) i2ret
INTEGER(4) i4ret
i4ret = SIGNALQQ(SIG$ABORT, h_abort)
WRITE(*,*) 'Set signal handler. Return = ', i4ret
i2ret = RAISEQQ(SIG$ABORT)
WRITE(*,*) 'Raised signal. Return = ', i2ret
END
!
!      Signal handler routine
!
INTEGER(4) FUNCTION h_abort (signum)
  !DEC$ ATTRIBUTES C :: h_abort
  INTEGER(2) signum
  WRITE(*,*) 'In signal handler for SIG$ABORT'
  WRITE(*,*) 'signum = ', signum
  h_abort = 1
END
```

See Also

- S
- RAISEQQ
- SIGNAL
- KILL
- GETEXCEPTIONPTRSQQ

SINGLE

OpenMP* Fortran Compiler Directive: *Specifies that a block of code is to be executed by only one thread in the team.*

Syntax

```
c$OMP SINGLE [clause[[,] clause] ... ]
   block
c$OMP END SINGLE [modifier]
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
clause	Is one of the following: • FIRSTPRIVATE (list) • PRIVATE (list)
block	Is a structured block (section) of statements or constructs. You cannot branch into or out of the block.
modifier	Is one of the following: • COPYPRIVATE (list) • NOWAIT

Threads in the team that are not executing this directive wait at the END SINGLE directive unless NOWAIT is specified.

SINGLE directives must be encountered by all threads in a team or by none at all. It must also be encountered in the same order by all threads in a team.

Example

In the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT:

```
c$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
c$OMP BARRIER
c$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
c$OMP END SINGLE
      CALL WORK(Y)
c$OMP END PARALLEL
```

You should not make assumptions as to which thread executes the SINGLE section. All other threads skip the SINGLE section and stop at the barrier at the END SINGLE construct. If other threads can proceed without waiting for the thread executing the SINGLE section, you can specify NOWAIT in the END SINGLE directive.

See Also

- S
- OpenMP Fortran Compiler Directives

SIZE Function

Inquiry Intrinsic Function (Generic): *Returns the total number of elements in an array, or the extent of an array along a specified dimension.*

Syntax

`result = SIZE (array [, dim] [, kind])`

array	(Input) Must be an array. It may be of any data type. It must not be a disassociated pointer or an allocatable array that is not allocated. It can be an assumed-size array if *dim* is present with a value less than the rank of *array*.

dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result is a scalar of type integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *dim* is present, the result is the extent of dimension *dim* in *array*; otherwise, the result is the total number of elements in *array*.

The setting of compiler options specifying integer size can affect this function.

Example

If B is declared as B(2:4, -3:1), then SIZE (B, DIM=2) has the value 5 and SIZE (B) has the value 15.

The following shows another example:

```
REAL(8) array (3:10, -1:3)
INTEGER i
i = SIZE(array, DIM = 2) ! returns 5
i = SIZE(array)          ! returns 40
```

See Also

- S
- SHAPE
- Character Count Specifier

SIZEOF

Inquiry Intrinsic Function (Generic): *Returns the number of bytes of storage used by the argument. It cannot be passed as an actual argument.*

Syntax

```
result = SIZEOF (x)
```

x	(Input) Can be a scalar or array. It may be of any data type. It must *not* be an assumed-size array.

Results

The result type is INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. The result value is the number of bytes of storage used by *x*.

Example

```
SIZEOF (3.44)              ! has the value 4
SIZEOF ('SIZE')            ! has the value 4
```

SLEEP

Portability Subroutine: *Suspends the execution of a process for a specified interval.*

Module

USE IFPORT

Syntax

```
CALL SLEEP (time)
```

time	(Input) INTEGER(4). Length of time, in seconds, to suspend the calling process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
integer(4) hold_time
hold_time = 1 !lets the loop execute
DO WHILE (hold_time .NE. 0)
  write(*,'(A)') "Enter the number of seconds to suspend"
  read(*,*) hold_time
  CALL SLEEP (hold_time)
END DO
END
```

See Also

- S
- SLEEPQQ

SLEEPQQ

Portability Subroutine: *Delays execution of the program for a specified duration.*

Module

USE IFPORT

Syntax

CALL SLEEPQQ (*duration*)

duration	(Input) INTEGER(4). Number of milliseconds the program is to sleep (delay program execution).

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) delay, freq, duration
delay    = 2000
freq     = 4000
duration = 1000
CALL SLEEPQQ(delay)
CALL BEEPQQ(freq, duration)
END
```

SNGL

See REAL function.

SORTQQ

Portability Subroutine: *Sorts a one-dimensional array. The array elements cannot be derived types or record structures.*

Module

USE IFPORT

Syntax

CALL SORTQQ (*adrarray*,*count*,*size*)

adrarray	(Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Address of the array (returned by LOC).
count	(Input; output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. On input, number of elements in the array to be sorted. On output, number of elements actually sorted. To be certain that SORTQQ is successful, compare the value returned in *count* to the value you provided. If they are the same, then SORTQQ sorted the correct number of elements.

size (Input) INTEGER(4). Positive constant less than 32,767 that specifies the kind of array to be sorted. The following constants, defined in IFPORT.F90, specify type and kind for numeric arrays:

Constant	Type of array
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) or equivalent
SRT$INTEGER4	INTEGER(4) or equivalent
SRT$INTEGER8	INTEGER(8) or equivalent
SRT$REAL4	REAL(4) or equivalent
SRT$REAL8	REAL(8) or equivalent
SRT$REAL16	REAL(16) or equivalent

If the value provided in *size* is not a symbolic constant and is less than 32,767, the array is assumed to be a character array with *size* characters per element.

CAUTION. The location of the array must be passed by address using the LOC function. This defeats Fortran type-checking, so you must make certain that the *count* and *size* arguments are correct.

If you pass invalid arguments, SORTQQ attempts to sort random parts of memory. If the memory it attempts to sort is allocated to the current process, that memory is sorted; otherwise, the operating system intervenes, the program is halted, and you get a General Protection Violation message.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!    Sort a 1-D array
!
USE IFPORT
INTEGER(2) array(10)
INTEGER(2) i
DATA ARRAY /143, 99, 612, 61, 712, 9112, 6, 555, 2223, 67/
!    Sort the array
Call SORTQQ (LOC(array), 10, SRT$INTEGER2)
!    Display the sorted array
DO i = 1, 10
  WRITE (*, 9000) i, array (i)
9000 FORMAT(1X, ' Array(',I2, '): ', I5)
END DO
END
```

See Also

- S
- BSEARCHQQ
- LOC

SPACING

Elemental Intrinsic Function (Generic):
Returns the absolute spacing of model numbers near the argument value.

Syntax

`result = SPACING (x)`

x	(Input) Must be of type real.

Results

The result type is the same as *x*. The result has the value b $^{e-p}$. Parameters *b*, *e*, and *p* are defined in Model for Real Data. If the result value is outside of the real model range, the result is TINY(*x*).

Example

If 3.0 is a REAL(4) value, SPACING (3.0) has the value 2 $^{-22}$.

The following shows another example:

```
REAL(4) res4
REAL(8) res8, r2
res4 = SPACING(3.0)   ! returns 2.384186E-07
res4 = SPACING(-3.0)  ! returns 2.384186E-07
r2   = 487923.3
res8 = SPACING(r2)    ! returns 5.820766091346741E-011
```

See Also

- S
- TINY
- RRSPACING
- Data Representation Models

SPLITPATHQQ

Portability Function: *Breaks a file path or directory path into its components.*

Module

USE IFPORT

Syntax

```
result = SPLITPATHQQ (path,drive,dir,name,ext)
```

path	(Input) Character*(*). Path to be broken into components. Forward slashes (/), backslashes (\), or both can be present in *path*.
drive	(Output) Character*(*). Drive letter followed by a colon.

dir	(Output) Character*(*). Path of directories, including the trailing slash.
name	(Output) Character*(*). Name of file or, if no file is specified in *path*, name of the lowest directory. A file name must not include an extension.
ext	(Output) Character*(*). File name extension, if any, including the leading period (.).

Results

The result type is INTEGER(4). The result is the length of *dir*.

The *path* parameter can be a complete or partial file specification.

$MAXPATH is a symbolic constant defined in module IFPORT.F90 as 260.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
CHARACTER($MAXPATH) buf
CHARACTER(3)        drive
CHARACTER(256)      dir
CHARACTER(256)      name
CHARACTER(256)      ext
CHARACTER(256)      file
INTEGER(4)          length
buf = 'b:\fortran\test\runtime\tsplit.for'
length = SPLITPATHQQ(buf, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
file = 'partial.f90'
length = SPLITPATHQQ(file, drive, dir, name, ext)
WRITE(*,*) drive, dir, name, ext
END
```

See Also

- S
- FULLPATHQQ

SPORT_CANCEL_IO (W*32, W*64)

Serial Port I/O Function: *Cancels any I/O in progress to the specified port.*

Module

USE IFPORT

Syntax

```
result = SPORT_CANCEL_IO (port)
```

port (Input) Integer. The port number.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. This call also kills the thread that keeps an outstanding read operation to the serial port. This call *must* be done before any of the port characteristics are modified.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult = SPORT_CANCEL_IO( 2 )

END
```

See Also

- S

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_CONNECT (W*32, W*64)

Serial Port I/O Function: *Establishes the connection to a serial port and defines certain usage parameters.*

Module

USE IFPORT

Syntax

`result = SPORT_CONNECT (`*`port`* `[,`*`options`*`])`

port (Input) Integer. The port number of connection. The routine will open COM *n*, where *n* is the port number specified.

options (Input; optional) Integer. Defines the connection options. These options define how the *nnn*_LINE routines will work and also effect the data that is passed to the user. If more than one option is specified, the operator .OR. should be used between each option. Options are as follows:

Option	Description
DL_TOSS_CR	Removes carriage return (CR) characters on input.
DL_TOSS_LF	Removes linefeed (LF) characters on input.
DL_OUT_CR	Causes SPORT_WRITE_LINE to add a CR to each record written.
DL_OUT_LF	Causes SPORT_WRITE_LINE to add a LF to each record written.
DL_TERM_CR	Causes SPORT_READ_LINE to terminate READ when a CR is encountered.

Option	Description
DL_TERM_LF	Causes SPORT_READ_LINE to terminate READ when a LF is encountered.
DL_TERM_CRLF	Causes SPORT_READ_LINE to terminate READ when CR+LF is encountered.

If *options* is not specified, the following occurs by default:

```
(DL_OUT_CR .OR. DL_TERM_CR .OR. DL_TOSS_CR .OR. DL_TOSS_LF)
```

This specifies to remove carriage returns and linefeeds on input, to follow output lines with a carriage return, and to return input lines when a carriage return is encountered.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_CONNECT( 2 )

END
```

See Also

- S
- SPORT_RELEASE, Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_CONNECT_EX (W*32, W*64)

Serial Port I/O Function: *Establishes the connection to a serial port, defines certain usage parameters, and defines the size of the internal buffer for data reception.*

Module

USE IFPORT

Syntax

`result = SPORT_CONNECT_EX (`*`port`* `[,`*`options`*`] [,`*`BufferSize`*`])`

port (Input) Integer. The port number of connection. The routine will open COM *n*, where *n* is the port number specified.

options (Input; optional) Integer. Defines the connection options. These options define how the *nnn*_LINE routines will work and also effect the data that is passed to the user. If more than one option is specified, the operator .OR. should be used between each option. Options are as follows:

Option	Description
DL_TOSS_CR	Removes carriage return (CR) characters on input.
DL_TOSS_LF	Removes linefeed (LF) characters on input.
DL_OUT_CR	Causes SPORT_WRITE_LINE to add a CR to each record written.
DL_OUT_LF	Causes SPORT_WRITE_LINE to add a LF to each record written.
DL_TERM_CR	Causes SPORT_READ_LINE to terminate READ when a CR is encountered.

Option	Description
DL_TERM_LF	Causes SPORT_READ_LINE to terminate READ when a LF is encountered.
DL_TERM_CRLF	Causes SPORT_READ_LINE to terminate READ when CR+LF is encountered.

If *options* is not specified, the following occurs by default:

```
(DL_OUT_CR .OR. DL_TERM_CR .OR. DL_TOSS_CR .OR. DL_TOSS_LF)
```

This specifies to remove carriage returns and linefeeds on input, to follow output lines with a carriage return, and to return input lines when a carriage return is encountered.

BufferSize (Input; optional) Integer. Size of the internal buffer for data reception. If *BufferSize* is not specified, the size of the buffer is 16384 bytes (the default).

The size of the buffer must be 4096 bytes or larger. If you try to specify a size smaller than 4096 bytes, your specification will be ignored and the buffer size will be set to 4096 bytes.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_CONNECT_EX( 2, BufferSize = 8196 )
END
```

See Also

- S

- SPORT_CONNECT
- SPORT_RELEASE

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_GET_HANDLE (W*32, W*64)

Serial Port I/O Function: *Returns the Windows* handle associated with the communications port. This is the handle that was returned by the Windows API CreateFile.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_HANDLE (port,handle)`

port	(Input) Integer. The port number.
handle	(Output) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. This is the Windows handle that was returned from CreatFile( ) on the serial port.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

INTEGER(KIND=INT_PTR_KIND( )) handle

iresult = SPORT_GET_HANDLE( 2, handle )

END
```

See Also

- S

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_GET_STATE (W*32, W*64)

Serial Port I/O Function: *Returns the baud rate, parity, data bits setting, and stop bits setting of the communications port.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_STATE (port [,baud] [,parity] [,dbits] [,sbits])`

port	(Input) Integer. The port number.
baud	(Output; optional) Integer. The baud rate of the port.
parity	(Output; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Output; optional) Integer. The data bits for the port.
sbits	(Output; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    baud
INTEGER    parity
INTEGER    dbits
INTEGER    sbits
iresult =  SPORT_GET_STATE( 2, baud, parity, dbits, sbits )
END
```

See Also

- S
- SPORT_SET_STATE

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_GET_STATE_EX (W*32, W*64)

Serial Port I/O Function: *Returns the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_GET_STATE_EX (port[,baud] [,parity] [,dbits] [,sbits] [,Binmode]
[,DTRcntrl]
   [,RTScntrl] [,OutCTSFlow] [,OutDSRFlow] [,DSRSense] [,OutXonOff] [,InXonOff]
[,XonLim]
   [,XoffLim] [,TXContOnXoff] [,ErrAbort] [,ErrCharEnbl] [,NullStrip]
[,XonChar] [,XoffChar]
   [,ErrChar] [,EofChar] [,EvtChar])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.
parity	(Output; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Output; optional) Integer. The data bits for the port.
sbits	(Output; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).
Binmode	(Output; optional) Integer. 1 if binary mode is enabled; otherwise, 0. Currently, the value of this parameter is always 1.
DTRcntrl	(Output; optional) Integer. 1 if DTR (data-terminal-ready) flow control is used; otherwise, 0.
RTScntrl	(Output; optional) Integer. 1 if RTS (request-to-send) flow control is used; otherwise, 0.
OutCTSFlow	(Output; optional) Integer. 1 if the CTS (clear-to-send) signal is monitored for output flow control; otherwise, 0.
OutDSRFlow	(Output; optional) Integer. 1 if the DSR (data-set-ready) signal is monitored for output flow control; otherwise, 0.
DSRSense	(Output; optional) Integer. 1 if the communications driver is sensitive to the state of the DSR signal; otherwise, 0.
OutXonOff	(Output; optional) Integer. 1 if XON/XOFF flow control is used during transmission; otherwise, 0.
InXonOff	(Output; optional) Integer. 1 if XON/XOFF flow control is used during reception; otherwise, 0.
XonLim	(Output; optional) Integer. The minimum number of bytes accepted in the input buffer before the XON character is set.
XoffLim	(Output; optional) Integer. The maximum number of bytes accepted in the input buffer before the XOFF character is set.
TXContOnXoff	(Output; optional) Integer. 1 if transmission stops when the input buffer is full and the driver has transmitted the *XoffChar* character; otherwise, 0.
ErrAbort	(Output; optional) Integer. 1 if read and write operations are terminated when an error occurs; otherwise, 0.
ErrCharEnbl	(Output; optional) Integer. 1 if bytes received with parity errors are replaced with the *ErrChar* character; otherwise, 0.

NullStrip	(Output; optional) Integer. 1 if null bytes are discarded; otherwise, 0.
XonChar	(Output; optional) Character. The value of the XON character that is used for both transmission and reception.
XoffChar	(Output; optional) Character. The value of the XOFF character that is used for both transmission and reception.
ErrChar	(Output; optional) Character. The value of the character that is used to replace bytes received with parity errors.
EofChar	(Output; optional) Character. The value of the character that is used to signal the end of data.
EvtChar	Output; optional) Character. The value of the character that is used to signal an event.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER(4) port, baud, parity, dbits, sbits
INTEGER(4) OutXonOff, InXonOff, OutDSRFlow
INTEGER(4) OutCTSFlow, DTRcntrl,  RTScntrl
INTEGER(4) DSRSense, XonLim, XoffLim
CHARACTER(1) XonChar, XoffChar
iresult = SPORT_GET_STATE_EX(port, baud, parity, dbits, sbits,               &
              OutXonOff=OutXonOff, InXonOff=InXonOff, OutDSRFlow=OutDSRFlow, &
              OutCTSFlow=OutCTSFlow, DTRcntrl=DTRcntrl, RTScntrl=RTScntrl,   &
              DSRSense = DSRSense, XonChar = XonChar, XoffChar = XoffChar,   &
              XonLim=XonLim, XoffLim=XoffLim)
END
```

See Also

- S
- SPORT_GET_STATE
- SPORT_SET_STATE_EX

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_GET_TIMEOUTS (W*32, W*64)

Serial Port I/O Function: *Returns the user selectable timeouts for the serial port.*

Module

USE IFPORT

Syntax

`result = SPORT_GET_TIMEOUTS (port [,rx_int] [,tx_tot_mult] [,tx_tot_const])`

port	(Input) Integer. The port number.
rx_int	(Output; optional) INTEGER(4). The receive interval timeout value.
tx_tot_mult	(Output; optional) INTEGER(4). The transmit multiplier part of the timeout value.
tx_tot_const	(Output; optional) INTEGER(4). The transmit constant part of the timeout value.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

INTEGER*4  rx_int

INTEGER*4  tx_tot_mult

INTEGER*4  tx_tot_const

iresult =  SPORT_GET_TIMEOUTS( 2, rx_int, tx_tot_mult, tx_tot_const )

END
```

See Also

- S
- SPORT_SET_TIMEOUTS

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_PEEK_DATA (W*32, W*64)

Serial Port I/O Function: *Returns information about the availability of input data.*

Module

USE IFPORT

Syntax

`result = SPORT_PEEK_DATA (port [,present] [,count])`

port	(Input) Integer. The port number.
present	(Output; optional) Integer. 1 if data is present, 0 if no data has been read.
count	(Output; optional) Integer. The count of characters that will be returned by SPORT_READ_DATA.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    present
INTEGER    count
iresult =  SPORT_PEEK_DATA( 2, present, count )
END
```

See Also

- S
- SPORT_CONNECT
- SPORT_READ_DATA
- SPORT_PEEK_LINE

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_PEEK_LINE (W*32, W*64)

Serial Port I/O Function: *Returns information about the availability of input records.*

Module

USE IFPORT

Syntax

```
result = SPORT_PEEK_LINE (port [,present] [,count])
```

port	(Input) Integer. The port number.
present	(Output; optional) Integer. 1 if data is present, 0 if no data has been read.
count	(Output; optional) Integer. The count of characters that will be returned by SPORT_READ_DATA.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the SPORT_CONNECT( ) call.

NOTE. CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
INTEGER    present
INTEGER    count
iresult =  SPORT_PEEK_LINE( 2, present, count )
END
```

See Also

- S
- SPORT_CONNECT
- SPORT_READ_DATA
- SPORT_PEEK_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_PURGE (W*32, W*64)

Serial Port I/O Function: *Executes the Windows* API communications function PurgeComm on the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_PURGE (port,function)`

port	(Input) Integer. The port number.
function	(Input) INTEGER(4). The function for PurgeComm (see the Windows documentation).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFWINTY

USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_PURGE( 2, (PURGE_TXABORT .or. PURGE_RXABORT) )

END
```

See Also

- S

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_READ_DATA (W*32, W*64)

Serial Port I/O Function: *Reads available data from the specified port. This routine stalls until at least one character has been read.*

Module

USE IFPORT

Syntax

`result = SPORT_READ_DATA (port,buffer[,count])`

port	(Input) Integer. The port number.
buffer	(Output) Character*(*). The data that was read.
count	(Output; optional) Integer. The count of bytes read.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4)      iresult
INTEGER         count
CHARACTER*1024  rbuff
iresult = SPORT_READ_DATA( 2, rbuff, count )
END
```

See Also

- S
- SPORT_CONNECT
- SPORT_PEEK_DATA
- SPORT_READ_LINE
- SPORT_WRITE_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_READ_LINE (W*32, W*64)

Serial Port I/O Function: *Reads a record from the specified port. This routine stalls until at least one record has been read.*

Module

USE IFPORT

Syntax

```
result = SPORT_READ_LINE (port,buffer[, count])
```

port	(Input) Integer. The port number.
buffer	(Output) Character*(*). The data that was read.
count	(Output; optional) Integer. The count of bytes read.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

This routine will only return when a line terminator has been seen - as defined by the mode specified in the SPORT_CONNECT( ) call.

NOTE. CR and LF characters may not be returned depending on the mode specified in the SPORT_CONNECT( ) call.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4)      iresult
INTEGER         count
CHARACTER*1024  rbuff
iresult = SPORT_READ_LINE( 2, rbuff, count )
END
```

See Also

- S
- SPORT_CONNECT
- SPORT_PEEK_LINE
- SPORT_READ_DATA
- SPORT_WRITE_LINE

Building Applications: Using the Serial I/O Port Routines

Communications and Communications Functions in the Microsoft* Platform SDK

SPORT_RELEASE (W*32, W*64)

Serial Port I/O Function: *Releases a serial port that was previously connected to (by using SPORT_CONNECT).*

Module

USE IFPORT

Syntax

`result = SPORT_RELEASE (port)`

port (Input) Integer. The port number.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_RELEASE( 2 )
END
```

See Also

- S
- SPORT_CONNECT

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_SET_STATE (W*32, W*64)

Serial Port I/O Function: *Sets the baud rate, parity, data bits setting, and stop bits setting of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_SET_STATE (port [,baud] [,parity] [,dbits] [,sbits])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.
parity	(Input; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Input; optional) Integer. The data bits for the port.
sbits	(Input; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

The following restrictions apply:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

NOTE. This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_SET_STATE( 2, 9600, 0, 7, 1 )

END
```

See Also

- S
- SPORT_CANCEL_IO
- SPORT_GET_STATE

Building Applications: Using the Serial I/O Port Routines

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

SPORT_SET_STATE_EX (W*32, W*64)

Serial Port I/O Function: *Sets the baud rate, parity, data bits setting, stop bits, and other settings of the communications port.*

Module

USE IFPORT

Syntax

```
result = SPORT_SET_STATE_EX (port [,baud] [,parity] [,dbits] [,sbits]
[,Binmode] [,DTRcntrl]
   [,RTScntrl] [,OutCTSFlow] [,OutDSRFlow] [,DSRSense] [,OutXonOff] [,InXonOff]
[,XonLim]
   [,XoffLim] [,TXContOnXoff] [,ErrAbort] [,ErrCharEnbl] [,NullStrip]
[,XonChar] [,XoffChar]
   [,ErrChar] [,EofChar] [,EvtChar] [,fZeroDCB])
```

port	(Input) Integer. The port number.
baud	(Input; optional) Integer. The baud rate of the port.

parity	(Input; optional) Integer. The parity setting of the port (0-4 = no, odd, even, mark, space).
dbits	(Input; optional) Integer. The data bits for the port.
sbits	(Input; optional) Integer. The stop bits for the port (0, 1, 2 = 1, 1.5, 2).
Binmode	(Input; optional) Integer. 1 if binary mode should be enabled; otherwise, 0. Currently, if this parameter is used, the value must be 1.
DTRcntrl	(Input; optional) Integer. 1 if DTR (data-terminal-ready) flow control should be used; otherwise, 0.
RTScntrl	(Input; optional) Integer. 1 if RTS (request-to-send) flow control should be used; otherwise, 0.
OutCTSFlow	(Input; optional) Integer. 1 if the CTS (clear-to-send) signal should be monitored for output flow control; otherwise, 0.
OutDSRFlow	(Input; optional) Integer. 1 if the DSR (data-set-ready) signal should be monitored for output flow control; otherwise, 0.
DSRSense	(Input; optional) Integer. 1 if the communications driver should be sensitive to the state of the DSR signal; otherwise, 0.
OutXonOff	(Input; optional) Integer. 1 if XON/XOFF flow control should be used during transmission; otherwise, 0.
InXonOff	(Input; optional) Integer. 1 if XON/XOFF flow control should be used during reception; otherwise, 0.
XonLim	(Input; optional) Integer. The minimum number of bytes that should be accepted in the input buffer before the XON character is set.
XoffLim	(Input; optional) Integer. The maximum number of bytes that should be accepted in the input buffer before the XOFF character is set.
TXContOnXoff	(Input; optional) Integer. 1 if transmission should be stopped when the input buffer is full and the driver has transmitted the *XoffChar* character; otherwise, 0.
ErrAbort	(Input; optional) Integer. 1 if read and write operations should be terminated when an error occurs; otherwise, 0.
ErrCharEnbl	(Input; optional) Integer. 1 if bytes received with parity errors should be replaced with the *ErrChar* character; otherwise, 0.

NullStrip	(Input; optional) Integer. 1 if null bytes should be discarded; otherwise, 0.
XonChar	(Input; optional) Character. The value of the XON character that should be used for both transmission and reception.
XoffChar	(Input; optional) Character. The value of the XOFF character that should be used for both transmission and reception.
ErrChar	(Input; optional) Character. The value of the character that should be used to replace bytes received with parity errors.
EofChar	(Input; optional) Character. The value of the character that should be used to signal the end of data.
EvtChar	(Input; optional) Character. The value of the character that should be used to signal an event.
fZeroDCB	(Input; optional) Integer. 1 if all settings of the communications port should be set to zero before parameters are set; otherwise, 0.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

The following restrictions apply:

- The number of data bits must be 5 to 8 bits.
- The use of 5 data bits with 2 stop bits is an invalid combination, as is 6, 7, or 8 data bits with 1.5 stop bits.

NOTE. This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_SET_STATE_EX( 2, 9600, 0, 7, 1, OutXonOff=1, InXonOff=1,     &
               XonLim=1024, XoffLim=512, XonChar=CHAR(17), XoffChar=CHAR(19), &
               fZeroDCB=1) )

END
```

See Also

- S
- SPORT_CANCEL_IO
- SPORT_GET_STATE
- SPORT_SET_STATE

Communications, Communications Functions, and SetCommState in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_SET_TIMEOUTS (W*32, W*64)

Serial Port I/O Function: *Sets the user selectable timeouts for the serial port.*

Module

USE IFPORT

Syntax

`result = SPORT_SET_TIMEOUTS (port [,rx_int] [,tx_tot_mult] [,tx_tot_const])`

port	(Input) Integer. The port number.
rx_int	(Input; optional) INTEGER(4). The receive interval timeout value.
tx_tot_mult	(Input; optional) INTEGER(4). The transmit multiplier part of the timeout value.
tx_tot_const	(Input; optional) INTEGER(4). The transmit constant part of the timeout value.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_SET_TIMEOUTS( 2, 100, 0, 1000 )

END
```

See Also

- S
- SPORT_CANCEL_IO
- SPORT_GET_TIMEOUTS

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_SHOW_STATE (W*32, W*64)

Serial Port I/O Function: *Displays the state of a port to standard output.*

Module

USE IFPORT

Syntax

`result = SPORT_SHOW_STATE (port,level)`

port (Input) Integer. The port number.

level (Input) Integer. Controls the level of detail displayed as follows:

0	Basic one line display
1	Basic information
2	Add modem signal control flow information
3	Add XON/XOFF information
4	Add event character information
11	Add timeout information
901	Add debug information

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. This routine must not be used when any I/O is pending. Since a read operation is always pending after any I/O has been started, you must first call SPORT_CANCEL_IO before port parameters can be changed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_SHOW_STATE( 2, 0 )
END
```

See Also

- S
- SPORT_CANCEL_IO

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_SPECIAL_FUNC (W*32, W*64)

Serial Port I/O Function: *Executes the Windows* API communications function EscapeCommFunction on the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_SPECIAL_FUNC (port,function)`

port	(Input) Integer. The port number.
function	(Input) INTEGER(4). The function to perform.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, the result is a Windows* error value.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult =  SPORT_SPECIAL_FUNC( 2, ? )

END
```

See Also

- S

SPORT_WRITE_DATA (W*32, W*64)

Serial Port I/O Function: *Outputs data to the specified port.*

Module

USE IFPORT

Syntax

`result = SPORT_WRITE_DATA (port,data[,count])`

port	(Input) Integer. The port number.
data	(Input) Character*(*). The data to be output.
count	(Input; optional) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

NOTE. When hardware (DTR, RTS, etc.) or software (XON/XOFF) flow controls are used, the functions SPORT_WRITE_DATA and SPORT_WRITE_LINE can write less bytes than required. When this occurs, the functions return the code ERROR_IO_INCOMPLETE, and the return value of parameter *count* contains the number of bytes that were really written.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) iresult
iresult =  SPORT_WRITE_DATA( 2, 'ATZ'//CHAR(13), 0 )
END
```

See Also

- S
- SPORT_WRITE_LINE
- SPORT_READ_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPORT_WRITE_LINE (W*32, W*64)

Serial Port I/O Function: *Outputs data, followed by a record terminator, to the specified port.*

Module

USE IFPORT

Syntax

```
result = SPORT_WRITE_LINE (port,data[,count])
```

port	(Input) Integer. The port number.
data	(Input) Character*(*). The data to be output.
count	(Input; optional) Integer. The count of bytes to write. If the value is zero, this number is computed by scanning the data backwards looking for a non-blank character.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, a Windows* error value.

After the data is output, a line terminator character is added based on the mode used during the SPORT_CONNECT( ) call.

NOTE. When hardware (DTR, RTS, etc.) or software (XON/XOFF) flow controls are used, the functions SPORT_WRITE_DATA and SPORT_WRITE_LINE can write less bytes than required. When this occurs, the functions return the code ERROR_IO_INCOMPLETE, and the return value of parameter *count* contains the number of bytes that were really written.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

INTEGER(4) iresult

iresult = SPORT_WRITE_LINE( 2, 'ATZ', 0 )

END
```

See Also

- S
- SPORT_CONNECT
- SPORT_WRITE_DATA
- SPORT_READ_DATA

Communications and Communications Functions in the Microsoft* Platform SDK

Building Applications: Using the Serial I/O Port Routines

SPREAD

Transformational Intrinsic Function (Generic):
Creates a replicated array with an added dimension by making copies of existing elements along a specified dimension.

Syntax

`result = SPREAD (source,dim,ncopies)`

source	(Input) Must be a scalar or array. It may be of any data type. The rank must be less than 7.

dim	(Input) Must be scalar and of type integer. It must have a value in the range 1 to n + 1 (inclusive), where *n* is the rank of *source*.
ncopies	Must be scalar and of type integer. It becomes the extent of the additional dimension in the result.

Results

The result is an array of the same type as *source* and of rank that is one greater than *source*.

If *source* is an array, each array element in dimension *dim* of the result is equal to the corresponding array element in *source*.

If *source* is a scalar, the result is a rank-one array with *ncopies* elements, each with the value *source*.

If *ncopies* less than or equal to zero, the result is an array of size zero.

Example

SPREAD ("B", 1, 4) is the character array (/"B", "B", "B", "B"/).

B is the array (3, 4, 5) and NC has the value 4.

SPREAD (B, DIM=1, NCOPIES=NC) produces the array

```
[ 3  4  5 ]
[ 3  4  5 ]
[ 3  4  5 ]
[ 3  4  5 ].
```

SPREAD (B, DIM=2, NCOPIES=NC) produces the array

```
[3  3  3  3 ]
[4  4  4  4 ]
[5  5  5  5 ].
```

The following shows another example:

```
INTEGER AR1(2, 3), AR2(3, 2)
AR1 = SPREAD((/1,2,3/),DIM= 1,NCOPIES= 2) ! returns
                                          ! 1 2 3
                                          ! 1 2 3
AR2 = SPREAD((/1,2,3/), 2, 2) ! returns   1 1
                              !           2 2
                              !           3 3
```

See Also

- S
- PACK
- RESHAPE

SQRT

Elemental Intrinsic Function (Generic):
Produces the square root of its argument.

Syntax

`result = SQRT (x)`

x	(Input) must be of type real or complex. If *x* is type real, its value must be greater than or equal to zero.

Results

The result type is the same as *x*. The result has a value equal to the square root of *x*. A result of type complex is the principal value, with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part of the result has the same sign as the imaginary part of *x* , even if the imaginary part of *x* is a negative real zero.

Specific Name	Argument Type	Result Type
SQRT	REAL(4)	REAL(4)
DSQRT	REAL(8)	REAL(8)
QSQRT	REAL(16)	REAL(16)

Specific Name	Argument Type	Result Type
CSQRT [1]	COMPLEX(4)	COMPLEX(4)
CDSQRT[2]	COMPLEX(8)	COMPLEX(8)
CQSQRT	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CSQRT.

[2]This function can also be specified as ZSQRT.

Example

SQRT (16.0) has the value 4.0.

SQRT (3.0) has the value 1.732051.

The following shows another example:

```
! Calculate the hypotenuse of a right triangle
! from the lengths of the other two sides.
REAL sidea, sideb, hyp
sidea = 3.0
sideb = 4.0
hyp = SQRT (sidea**2 + sideb**2)
WRITE (*, 100) hyp
100 FORMAT (/ ' The hypotenuse is ', F10.3)
END
```

SRAND

Portability Subroutine: *Seeds the random number generator used with IRAND and RAND.*

Module

USE IFPORT

Syntax

CALL SRAND (*iseed*)

iseed (Input) INTEGER(4). Any value. The default value is 1.

SRAND seeds the random number generator used with IRAND and RAND. Calling SRAND is equivalent to calling IRAND or RAND with a new seed.

The same value for *iseed* generates the same sequence of random numbers. To vary the sequence, call SRAND with a different *iseed* value each time the program is executed.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
!  How many random numbers out of 100 will be between .5 and .6?
   USE IFPORT
   ICOUNT = 0
   CALL SRAND(123)
   DO I = 1, 100
     X = RAND(0)
     IF ((X>.5).AND.(x<.6)) ICOUNT = ICOUNT + 1
   END DO
   WRITE(*,*) ICOUNT, "numbers between .5 and .6!"
   END
```

See Also

- S
- RAND
- IRAND
- RANDOM_NUMBER
- RANDOM_SEED

SSWRQQ

Portability Subroutine: *Returns the floating-point processor status word.*

Module

USE IFPORT

Syntax

`CALL SSWRQQ (status)`

status (Output) INTEGER(2). Floating-point processor status word.

SSWRQQ performs the same function as the run-time subroutine GETSTATUSFPQQ and is provided for compatibility.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(2) status
CALL SSWRQQ (status)
```

See Also

- S
- LCWRQQ
- GETSTATUSFPQQ

STAT

Portability Function: *Returns detailed information about a file.*

Module

USE IFPORT

Syntax

```
result = STAT (name,statb)
```

name (Input) Character*(*). Name of the file to examine.

statb (Output) INTEGER(4) or INTEGER(8). One-dimensional array of size 12; where the system information is stored. The elements of *statb* contain the following values:

Element	Description	Values or Notes
statb(1)	Device the file resides on	W*32, W*64: Always 0 L*X, M*X: System dependent
statb(2)	File inode number	W*32, W*64: Always 0 L*X, M*X: System dependent
statb(3)	Access mode of the file	See the table in Results
statb(4)	Number of hard links to the file	W*32, W*64: Always 1 L*X, M*X: System dependent
statb(5)	User ID of owner	W*32, W*64: Always 1 L*X, M*X: System dependent
statb(6)	Group ID of owner	W*32, W*64: Always 1 L*X, M*X: System dependent

Element	Description	Values or Notes
statb(7)	Raw device the file resides on	W*32, W*64: Always 0 L*X, M*X: System dependent
statb(8)	Size of the file	
statb(9)	Time when the file was last accessed[1]	W*32, W*64: Only available on non-FAT file systems; undefined on FAT systems L*X, M*X: System dependent
statb(10)	Time when the file was last modified[1]	
statb(11)	Time of last file status change[1]	W*32, W*64: Same as stat(10) L*X, M*X: System dependent
statb(12)	Blocksize for file system I/O operations	W*32, W*64: Always 1 L*X, M*X: System dependent

[1]Times are in the same format returned by the TIME function (number of seconds since 00:00:00 Greenwich mean time, January 1, 1970).

Results

The result type is INTEGER(4).

On Windows* systems, the result is zero if the inquiry was successful; otherwise, the error code ENOENT (the specified file could not be found). On Linux* and Mac OS* X systems, the file inquired about must be currently connected to a logical unit and must already exist when STAT is called; if STAT fails, `errno`is set.

For a list of other error codes, see IERRNO.

The access mode (the third element of *statb*) is a bitmap consisting of an IOR of the following constants:

Symbolic name	Constant	Description	Notes
S_IFMT	O'0170000'	Type of file	
S_IFDIR	O'0040000'	Directory	
S_IFCHR	O'0020000'	Character special	Never set on Windows systems
S_IFBLK	O'0060000'	Block special	Never set on Windows systems
S_IFREG	O'0100000'	Regular	
S_IFLNK	O'0120000'	Symbolic link	Never set on Windows systems
S_IFSOCK	O'0140000'	Socket	Never set on Windows systems
S_ISUID	O'0004000'	Set user ID on execution	Never set on Windows systems
S_ISGID	O'0002000'	Set group ID on execution	Never set on Windows systems
S_ISVTX	O'0001000'	Save swapped text	Never set on Windows systems
S_IRWXU	O'0000700'	Owner's file permissions	

Symbolic name	Constant	Description	Notes
S_IRUSR, S_IREAD	O'0000400'	Owner's read permission	Always true on Windows systems
S_IWUSR, S_IWRITE	O'0000200'	Owner's write permission	
S_IXUSR, S_IEXEC	O'0000100'	Owner's execute permission	Based on file extension (.EXE, .COM, .CMD, or .BAT)
S_IRWXG	O'0000070'	Group's file permissions	Same as S_IRWXU on Windows systems
S_IRGRP	O'0000040'	Group's read permission	Same as S_IRUSR on Windows systems
S_IWGRP	O'0000020'	Group's write permission	Same as S_IWUSR on Windows systems
S_IXGRP	O'0000010'	Group's execute permission	Same as S_IXUSR on Windows systems
S_IRWXO	O'0000007'	Other's file permissions	Same as S_IRWXU on Windows systems
S_IROTH	O'0000004'	Other's read permission	Same as S_IRUSR on Windows systems
S_IWOTH	O'0000002'	Other's write permission	Same as S_IWUSR on Windows systems
S_IXOTH	O'0000001'	Other's execute permission	Same as S_IXUSR on Windows systems

STAT returns the same information as FSTAT, but accesses files by name instead of external unit number.

On Windows systems, LSTAT returns exactly the same information as STAT. On Linux and Mac OS X systems, if the file denoted by *name* is a link, LSTAT provides information on the link, while STAT provides information on the file at the destination of the link.

You can also use the INQUIRE statement to get information about file properties.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
CHARACTER*12 file_name
INTEGER(4) info_array(12)
print *, 'Enter file to examine: '
read *, file_name
ISTATUS = STAT (file_name, info_array)
if (.not. istatus) then
  print *, info_array
else
  print *, 'Error = ',istatus
end if
end
```

See Also

- S
- INQUIRE
- GETFILEINFOQQ
- FSTAT

Statement Function

Statement: *Defines a function in a single statement in the same program unit in which the procedure is referenced.*

Syntax

fun([*d-arg* [,*d-arg*]...]) = *expr*

fun	Is the name of the statement function.

d-arg	Is a dummy argument. A dummy argument can appear only once in any list of dummy arguments, and its scope is local to the statement function.
expr	Is a scalar expression defining the computation to be performed. Named constants and variables used in the expression must have been declared previously in the specification part of the scoping unit or made accessible by use or host association. If the expression contains a function or statement function reference, that function must have been defined previously as a function or statement function in the same program unit. A statement function reference takes the following form: *fun*([*a-arg*[, *a-arg*] ...])
fun	Is the name of the statement function.
a-arg	Is an actual argument.

Description

When a statement function reference appears in an expression, the values of the actual arguments are associated with the dummy arguments in the statement function definition. The expression in the definition is then evaluated. The resulting value is used to complete the evaluation of the expression containing the function reference.

The data type of a statement function can be explicitly defined in a type declaration statement. If no type is specified, the type is determined by implicit typing rules in effect for the program unit.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments.

Except for the data type, declarative information associated with an entity is not associated with dummy arguments in the statement function; for example, declaring an entity to be an array or to be in a common block does not affect a dummy argument with the same name.

The name of the statement function cannot be the same as the name of any other entity within the same program unit.

Any reference to a statement function must appear in the same program unit as the definition of that function.

A statement function reference must appear as (or be part of) an expression. The reference cannot appear on the left side of an assignment statement.

A statement function must not be provided as a procedure argument.

Example

The following are examples of statement functions:

```
REAL VOLUME, RADIUS

VOLUME(RADIUS) = 4.189*RADIUS**3

CHARACTER*10 CSF,A,B

CSF(A,B) = A(6:10)//B(1:5)
```

The following example shows a statement function and some references to it:

```
AVG(A,B,C) = (A+B+C)/3.

...

GRADE = AVG(TEST1,TEST2,XLAB)

IF (AVG(P,D,Q) .LT. AVG(X,Y,Z)) STOP

FINAL = AVG(TEST3,TEST4,LAB2)       ! Invalid reference; implicit

...                                 ! type of third argument does not

...                                 ! match implicit type of dummy argument
```

Implicit typing problems can be avoided if all arguments are explicitly typed.

The following statement function definition is invalid because it contains a constant, which cannot be used as a dummy argument:

```
REAL COMP, C, D, E

COMP(C,D,E,3.) = (C + D - E)/3.
```

The following shows another example:

```
Add (a, b) = a + b

REAL(4) y, x(6)

. . .

DO n = 2, 6

  x(n) = Add (y, x(n-1))

END DO
```

See Also

- S
- FUNCTION

- Argument Association
- Use and Host Association

STATIC

Statement and Attribute: *Controls the storage allocation of variables in subprograms (as does AUTOMATIC). Variables declared as STATIC and allocated in memory reside in the static storage area, rather than in the stack storage area.*

Syntax

The STATIC attribute can be specified in a type declaration statement or a STATIC statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls,] STATIC [, att-ls] :: v[, v] ...
```

Statement:

```
STATIC[::] v[, v] ...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
v	Is the name of a variable or an array specification. It can be of any type.

STATIC declarations only affect how data is allocated in storage.

If you want to retain definitions of variables upon reentry to subprograms, you must use the SAVE attribute.

By default, the compiler allocates local scalar variables of non-recursive subprograms in the static storage area. Local arrays, except for allocatable arrays, are in the static storage area by default.

The compiler may choose to allocate a variable in temporary (stack or register) storage if it notices that the variable is always defined before use. Appropriate use of the SAVE attribute can prevent compiler warnings if a variable is used before it is defined.

To change the default for variables, specify them as AUTOMATIC or specify RECURSIVE in one of the following ways:

- As a keyword in a FUNCTION or SUBROUTINE statement
- As a compiler option

- As an option in an OPTIONS statement

To override any compiler option that may affect variables, explicitly specify the variables as STATIC.

NOTE. Variables that are data-initialized, and variables in COMMON and SAVE statements are always static. This is regardless of whether a compiler option specifies recursion.

A variable cannot be specified as STATIC more than once in the same scoping unit.

If the variable is a pointer, STATIC applies only to the pointer itself, not to any associated target.

Some variables cannot be specified as STATIC. The following table shows these restrictions:

Variable	STATIC
Dummy argument	No
Automatic object	No
Common block item	Yes
Use-associated item	No
Function result	No
Component of a derived type	No

A variable can be specified with both the STATIC and SAVE attributes.

If a variable is in a module's outer scope, it can be specified as STATIC.

Example

The following example shows a type declaration statement specifying the STATIC attribute:

```
INTEGER, STATIC :: ARRAY_A
```

The following example uses a STATIC statement:

```
...
CONTAINS
 INTEGER FUNCTION REDO_FUNC
   INTEGER I, J(10), K
   REAL C, D, E(30)
   AUTOMATIC I, J, K(20)
   STATIC C, D, E
   ...
 END FUNCTION
...
 INTEGER N1, N2
 N1 = -1
 DO WHILE (N1)
   N2 = N1*2
   call sub1(N1, N2)
   read *, N1
 END DO
 CONTAINS
 SUBROUTINE sub1 (iold, inew)
 INTEGER, intent(INOUT):: iold
 integer, STATIC ::N3
 integer, intent(IN) :: inew
 if (iold .eq. -1) then
   N3 = iold
 end if
 print *, 'New: ', inew, 'N3: ',N3
 END subroutine
 !
```

```
END
```

See Also

- S
- AUTOMATIC
- SAVE
- Type declaration statements
- Compatible attributes
- RECURSIVE
- OPTIONS
- POINTER
- Modules and Module Procedures
- recursive compiler option

STOP

Statement: *Terminates program execution before the end of the program unit.*

Syntax

`STOP [stop-code]`

stop-code	(Optional) A message. It can be either of the following: • A scalar character constant of type default character. • A non-negative integer less than or equal to 2147483647; leading zeros are ignored. (The Fortran standard limits *stop-code* to at most five digits.)

If *stop-code* is specified, the STOP statement does the following:

- Writes the specified message to the standard error device.
- Writes one or more of the following messages to the standard error device indicating which IEEE floating-point exceptions are signaling:
  - IEEE_DIVIDE_BY_ZERO is signaling
  - IEEE_INVALID is signaling
  - IEEE_OVERFLOW is signaling

  - IEEE_UNDERFLOW is signaling

- Terminates program execution. If *stop-code* is a character constant, a status of zero is returned. If *stop-code* is an integer, a status equal to *stop-code* is returned.

If *stop-code* is not specified, the program is terminated, no message is printed, and a status of zero is returned.

Effect on Windows* Systems

In QuickWin programs, the following is displayed in a message box:

```
Program terminated with Exit Code stop-code
```

Effect on Linux* and Mac OS* Systems

Operating system shells (such as bash, sh, csh, etc.) work with one byte exit status. So, when *stop-code* is an integer, only the lowest byte is significant. For example, consider the following statement:

```
STOP 99999
```

In this case, the program returns a status equal to 159 because integer 99999 = Z'1869F', and the lowest byte is equal to Z'9F', which equals 159.

Example

The following examples show valid STOP statements:

```
STOP 98

STOP 'END OF RUN'

DO

  READ *, X, Y

  IF (X > Y) STOP 5555

END DO
```

The following shows another example:

```
      OPEN(1,FILE='file1.dat', status='OLD', ERR=100)

      . . .

 100  STOP 'ERROR DETECTED!'

      END
```

See Also

- S
- EXIT

STRICT and NOSTRICT

General Compiler Directive: *STRICT disables language features not found in the language standard specified on the command line (Fortran 2003, Fortran 95, or Fortran 90). NOSTRICT (the default) enables these features.*

Syntax

```
cDEC$ STRICT
cDEC$ NOSTRICT
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

If STRICT is specified and no language standard is specified on the command line, the default is to disable features not found in Fortran 2003.

The STRICT and NOSTRICT directives can appear only appear at the top of a program unit. A program unit is a main program, an external subroutine or function, a module, or a block data program unit. STRICT and NOSTRICT cannot appear between program units, or at the beginning of internal subprograms. They do not affect any modules invoked with the USE statement in the program unit that contains them.

Example

```
! NOSTRICT by default
TYPE stuff
  INTEGER(4) k
  INTEGER(4) m
  CHARACTER(4) name
END TYPE stuff
TYPE (stuff) examp
DOUBLE COMPLEX cd  ! non-standard data type, no error
cd =(3.0D0, 4.0D0)
examp.k = 4        ! non-standard component designation,
                   ! no error
END
SUBROUTINE STRICTDEMO( )
  !DEC$ STRICT
   TYPE stuff
    INTEGER(4) k
    INTEGER(4) m
    CHARACTER(4) name
   END TYPE stuff
   TYPE (stuff) samp
   DOUBLE COMPLEX cd   ! ERROR
   cd =(3.0D0, 4.0D0)
   samp.k = 4          ! ERROR
END SUBROUTINE
```

See Also

- M to N
- S

- General Compiler Directives
- stand compiler option

Building Applications: Compiler Directives Related to Options

STRUCTURE...END STRUCTURE

Statement: *Defines the field names, types of data within fields, and order and alignment of fields within a record structure. Fields and structures can be initialized, but records cannot be initialized.*

Syntax

```
STRUCTURE [/structure-name/] [field-namelist]
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END STRUCTURE
```

structure-name	Is the name used to identify a structure, enclosed by slashes. Subsequent RECORD statements use the structure name to refer to the structure. A structure name must be unique among structure names, but structures can share names with variables (scalar or array), record fields, PARAMETER constants, and common blocks. Structure declarations can be nested (contain one or more other structure declarations). A structure name is required for the structured declaration at the outermost level of nesting, and is optional for the other declarations nested in it. However, if you wish to reference a nested structure in a RECORD statement in your program, it must have a name. Structure, field, and record names are all local to the defining program unit. When records are passed as arguments, the fields in the defining structures within the calling and called subprograms must match in type, order, and dimension.
field-namelist	Is a list of fields having the structure of the associated structure declaration. A field namelist is allowed only in nested structure declarations.

field-declaration

Also called the declaration body. A *field-declaration* consists of any combination of the following:

- Type declarations

  These are ordinary Fortran data type declarations.

- Substructure declarations

  A field within a structure can be a substructure composed of atomic fields, other substructures, or a combination of both.

- Union declarations

  A union declaration is composed of one or more mapped field declarations.

- PARAMETER statements

  PARAMETER statements can appear in a structure declaration, but cannot be given a data type within the declaration block.

  Type declarations for PARAMETER names must precede the PARAMETER statement and be outside of a STRUCTURE declaration, as follows:

```
   INTEGER*4 P
 STRUCTURE /ABC/
   PARAMETER (P=4)
   REAL*4 F
 END STRUCTURE
   REAL*4 A(P)
```

The Fortran 90 derived type replaces STRUCTURE and RECORD constructs, and should be used in writing new code. See Derived Data Types.

Unlike type declaration statements, structure declarations do not create variables. Structured variables (records) are created when you use a RECORD statement containing the name of a previously declared structure. The RECORD statement can be considered as a kind of type declaration statement. The difference is that aggregate items, not single items, are being defined.

Within a structure declaration, the ordering of both the statements and the field names within the statements is important, because this ordering determines the order of the fields in records.

In a structure declaration, each field offset is the sum of the lengths of the previous fields, so the length of the structure is the sum of the lengths of its fields. The structure is packed; you must explicitly provide any alignment that is needed by including, for example, unnamed fields of the appropriate length.

By default, fields are aligned on natural boundaries; misaligned fields are padded as necessary. To avoid padding of records, you should lay out structures so that all fields are naturally aligned.

To pack fields on arbitrary byte boundaries, you must specify a compiler option. You can also specify alignment for fields by using the OPTIONS or PACK general directive.

A field name must not be the same as any intrinsic or user-defined operator (for example, EQ cannot be used as a field name).

Compatibility

An item can be a RECORD statement that references a previously defined structure type:

```
STRUCTURE /full_address/
  RECORD  /full_name/ personsname
  RECORD  /address/   ship_to
  INTEGER*1           age
  INTEGER*4           phone
END STRUCTURE
```

You can specify a particular item by listing the sequence of items required to reach it, separated by a period (.). Suppose you declare a structure variable, `shippingaddress`, using the `full_address`structure defined in the previous example:

```
RECORD /full_address/ shippingaddress
```

In this case, the `age`item would then be specified by `shippingaddress.age`, the first name of the receiver by `shippingaddress.personsname.first_name`, and so on.

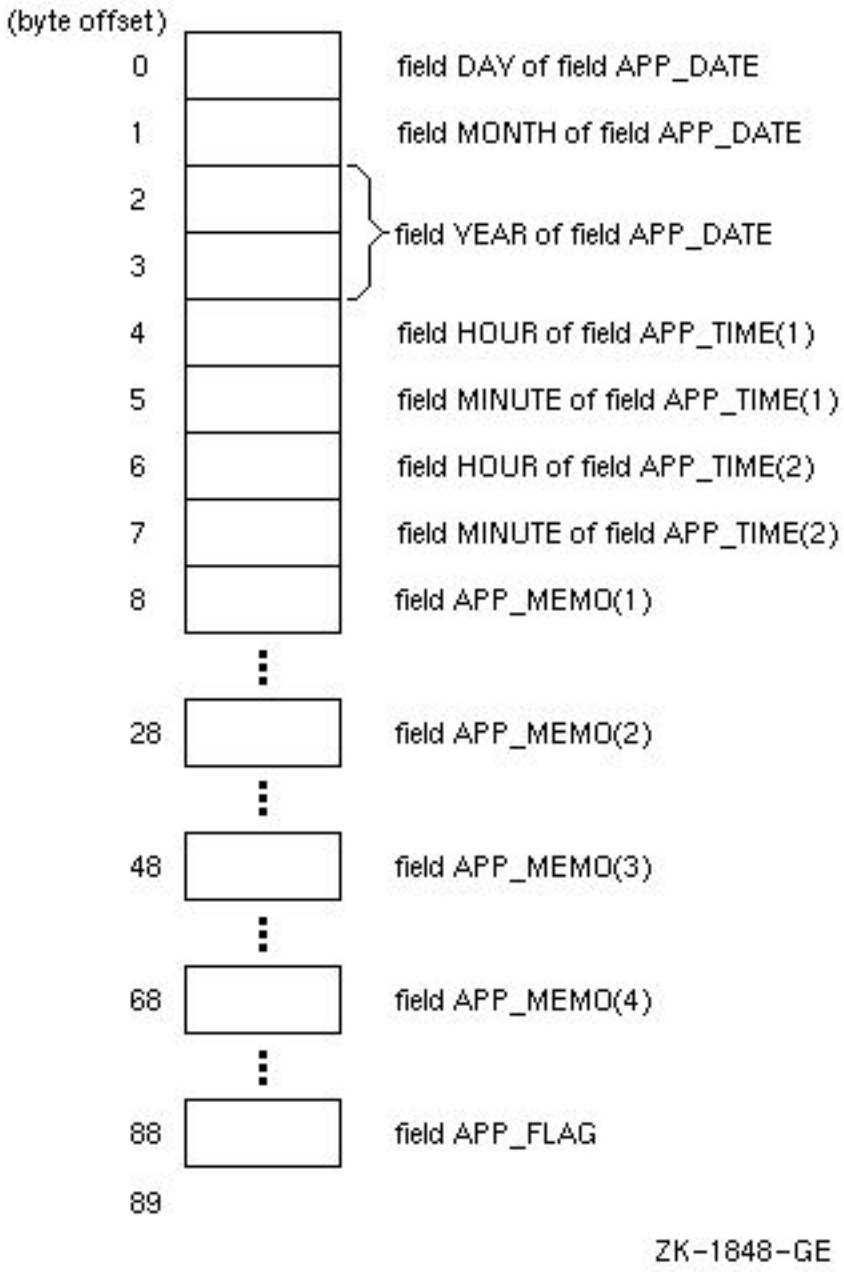
In the following example, the declaration defines a structure named APPOINTMENT. APPOINTMENTcontains the structure DATE(field APP_DATE) as a substructure. It also contains a substructure named TIME(field APP_TIME, an array), a CHARACTER*20 array named APP_MEMO, and a LOGICAL*1 field named APP_FLAG.

```
STRUCTURE /DATE/
  INTEGER*1 DAY, MONTH
  INTEGER*2 YEAR
END STRUCTURE
STRUCTURE /APPOINTMENT/
  RECORD /DATE/     APP_DATE
  STRUCTURE /TIME/  APP_TIME (2)
    INTEGER*1       HOUR, MINUTE
  END STRUCTURE
  CHARACTER*20      APP_MEMO (4)
  LOGICAL*1         APP_FLAG
END STRUCTURE
```

The length of any instance of structure APPOINTMENT is 89 bytes.

The following figure shows the memory mapping of any record or record array element with the structure `APPOINTMENT`.

Figure 100: Memory Map of Structure APPOINTMENT


(byte offset)
0
1
2
3
4
5
6
7
8
28
48
68
88
89
field DAY of field APP_DATE
field MONTH of field APP_DATE
field YEAR of field APP_DATE
field HOUR of field APP_TIME(1)
field MINUTE of field APP_TIME(1)
field HOUR of field APP_TIME(2)
field MINUTE of field APP_TIME(2)
field APP_MEMO(1)
field APP_MEMO(2)
field APP_MEMO(3)
field APP_MEMO(4)
field APP_FLAG
ZK-1848-GE

See Also

- E to F
- S
- TYPE
- MAP...END MAP
- RECORD
- UNION...END UNION
- PACK Directive
- OPTIONS Directive
- Data Types, Constants, and Variables
- Record Structures

SUBROUTINE

Statement: *The initial statement of a subroutine subprogram. A subroutine subprogram is invoked in a CALL statement or by a defined assignment statement, and does not return a particular value.*

Syntax

```
[prefix [prefix]] SUBROUTINE name [([d-arg-list]) [lang-binding]]
   [specification-part]
   [execution-part]
[CONTAINS
   internal-subprogram-part]
END [SUBROUTINE [name]]
```

prefix (Optional) Is any of the following:

- RECURSIVE

  Permits direct recursion to occur.
- PURE

  Asserts that the procedure has no side effects.
- ELEMENTAL

Acts on one array element at a time. This is a restricted form of pure procedure.

At most one of each of the above can be specified. You cannot specify ELEMENTAL and RECURSIVE together. You cannot specify ELEMENTAL if *lang-binding* is specified.

name Is the name of the subroutine.

d-arg-list (Optional) Is a list of one or more dummy arguments or alternate return specifiers (*) .

lang-binding (Optional) Takes the following form:
BIND (C [, NAME=*ext-name*])

ext-name Is a character scalar initialization expression that can be used to construct the external name.

specification-part Is one or more specification statements, except for the following:

- INTENT (or its equivalent attribute)
- OPTIONAL (or its equivalent attribute)
- PUBLIC and PRIVATE (or their equivalent attributes)

An automatic object must not appear in a specification statement. If a SAVE statement is specified, it has no effect.

execution-part Is one or more executable constructs or statements, except for ENTRY or RETURN statements.

internal-subprogram-part Is one or more internal subprograms (defining internal procedures). The *internal-subprogram-part* is preceded by a CONTAINS statement.

Description

A subroutine is invoked by a CALL statement or defined assignment. When a subroutine is invoked, dummy arguments (if present) become associated with the corresponding actual arguments specified in the call.

Execution begins with the first executable construct or statement following the SUBROUTINE statement. Control returns to the calling program unit once the END statement (or a RETURN statement) is executed.

A subroutine subprogram *cannot* contain a BLOCK DATA statement, a PROGRAM statement, or a MODULE statement. A subroutine can contain SUBROUTINE and FUNCTION statements to define internal procedures. ENTRY statements can be included to provide multiple entry points to the subprogram.

You need an interface block for a subroutine when:

- Calling arguments use argument keywords.
- Some arguments are optional.
- A dummy argument is an assumed-shape array, a pointer, or a target.
- The subroutine extends intrinsic assignment.
- The subroutine can be referenced by a generic name.
- The subroutine is in a dynamic-link library.

If the subroutine is in a DLL and is called from your program, use the option DLLEXPORT or DLLIMPORT, which you can specify with the ATTRIBUTES directive.

Note that if you specify *lang-binding*, you have to use the parentheses even if there are no arguments. For example, without *lang-binding* you can specify `SUBROUTINE F` but with *lang-binding* you have to specify `SUBROUTINE F( ) BIND (C)`.

Example

The following example shows a subroutine:

```
Main Program          Subroutine
CALL HELLO_WORLD      SUBROUTINE HELLO_WORLD
...                   PRINT *, "Hello World"
END                   END SUBROUTINE
```

The following example uses alternate return specifiers to determine where control transfers on completion of the subroutine:

Main Program

```
    CALL CHECK(A,B,*10,*20,C)
    TYPE *, 'VALUE LESS THAN ZERO'
    GO TO 30
10  TYPE*, 'VALUE EQUALS ZERO'
    GO TO 30
20  TYPE*, 'VALUE MORE THAN ZERO'
30  CONTINUE
    ...
```

Subroutine

```
    SUBROUTINE CHECK(X,Y,*,*,Q)
    ...
50  IF (Z)  60,70,80
60  RETURN
70  RETURN 1
80  RETURN 2
    END
```

The SUBROUTINE statement argument list contains two dummy alternate return arguments corresponding to the actual arguments *10 and *20 in the CALL statement argument list.

The value of Z determines the return, as follows:

- If Z < zero, a normal return occurs and control is transferred to the first executable statement following CALL CHECK in the main program.
- If Z = = zero, the return is to statement label 10 in the main program.
- If Z > zero, the return is to statement label 20 in the main program.

(An alternate return is an obsolescent feature in Fortran 90 and Fortran 95.)

The following shows another example:

```
     SUBROUTINE GetNum (num, unit)
     INTEGER num, unit
 10  READ (unit, '(I10)', ERR = 10) num
     END
```

See Also

- S
- FUNCTION
- INTERFACE
- PURE

- ELEMENTAL
- CALL
- RETURN
- ENTRY
- Argument Association
- Program Units and Procedures
- General Rules for Function and Subroutine Subprograms
- Obsolescent and Deleted Language Features

SUM

Transformational Intrinsic Function (Generic):
Returns the sum of all the elements in an entire array or in a specified dimension of an array.

Syntax

```
result = SUM (array [,dim] [,mask])
```

array	(Input) Must be an array of type integer, real, or complex.
dim	(Input; optional) Must be a scalar integer with a value in the range 1 to n, where *n* is the rank of *array*.
mask	(Input; optional) Must be of type logical and conformable with *array*.

Results

The result is an array or a scalar of the same data type as *array*.

The result is a scalar if *dim* is omitted or *array* has rank one.

The following rules apply if *dim* is omitted:

- If SUM(*array*) is specified, the result is the sum of all elements of *array*. If *array* has size zero, the result is zero.
- If SUM(*array*, MASK= *mask*) is specified, the result is the sum of all elements of *array* corresponding to true elements of *mask*. If *array* has size zero, or every element of *mask* has the value .FALSE., the result is zero.

The following rules apply if *dim* is specified:

- If *array* has rank one, the value is the same as SUM(*array*[,MASK= *mask*]).

- An array result has a rank that is one less than *array*, and shape (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), where (d_1, d_2, ..., d_n) is the shape of *array*.
- The value of element (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) of SUM(*array*, *dim*[, *mask*]) is equal to SUM(*array*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n) [,MASK = *mask*(s_1, s_2, ..., s_{dim-1}, :, s_{dim+1}, ..., s_n)].

Example

SUM ((/2, 3, 4/)) returns the value 9 (sum of 2 + 3 + 4). SUM ((/2, 3, 4/), DIM=1) returns the same result.

SUM (B, MASK=B .LT. 0.0) returns the arithmetic sum of the negative elements of B.

C is the array

```
[ 1  2  3 ]
[ 4  5  6 ].
```

SUM (C, DIM=1) returns the value (5, 7, 9), which is the sum of all elements in each column. 5 is the sum of 1 + 4 in column 1. 7 is the sum of 2 + 5 in column 2, and so forth.

SUM (C, DIM=2) returns the value (6, 15), which is the sum of all elements in each row. 6 is the sum of 1 + 2 + 3 in row 1. 15 is the sum of 4 + 5 + 6 in row 2.

The following shows another example:

```
INTEGER array (2, 3), i, j(3)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
!  array is  1 3 5
!            2 4 6
i = SUM((/ 1, 2, 3 /))        ! returns 6
j = SUM(array, DIM = 1)       ! returns [3 7 11]
WRITE(*,*) i, j
END
```

See Also

- S
- PRODUCT

SWP and NOSWP (i64 only)

General Compiler Directives: *SWP enables software pipelining for a DO loop. NOSWP (the default) disables this software pipelining. These directives are only available on IA-64 architecture.*

Syntax

*c*DEC$ SWP

*c*DEC$ NOSWP

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The SWP directive must precede the DO statement for each DO loop it affects.

The SWP directive does not help data dependence, but overrides heuristics based on profile counts or lop-sided control flow.

The software pipelining optimization specified by the SWP directive applies instruction scheduling to certain innermost loops, allowing instructions within a loop to be split into different stages.

This allows increased instruction level parallelism, which can reduce the impact of long-latency operations, resulting in faster loop execution.

Loops chosen for software pipelining are always innermost loops containing procedure calls that are inlined. Because the optimizer no longer considers fully unrolled loops as innermost loops, fully unrolling loops can allow an additional loop to become the innermost loop (see compiler option -funroll-loops or /Qunroll).

You can request and view the optimization report to see whether software pipelining was applied.

Example

```
!DEC$ SWP
  do i = 1, m
    if (a(i) .eq. 0) then
      b(i) = a(i) + 1
    else
      b(i) = a(i)/c(i)
    endif
  enddo
```

See Also

- M to N
- S
- Syntax Rules for Compiler Directives
- Rules for General Directives that Affect DO Loops
- unroll, Qunroll compiler option

Optimizing Applications: Pipelining for Itanium®-based Architecture

Optimizing Applications: Optimizer Report Generation

SYSTEM

Portability Function: *Sends a command to the shell as if it had been typed at the command line.*

Module

USE IFPORT

Syntax

`result = SYSTEM (string)`

string	(Input) Character*(*). Operating system command.

Results

The result type is INTEGER(4). The result is the exit status of the shell command. If -1, use IERRNO to retrieve the error. Errors can be one of the following:

- E2BIG: The argument list is too long.
- ENOENT: The command interpreter cannot be found.
- ENOEXEC: The command interpreter file has an invalid format and is not executable.
- ENOMEM: Not enough system resources are available to execute the command.

On Windows* systems, the calling process waits until the command terminates. To insure compatibility and consistent behavior, an image can be invoked directly by using the Windows API CreateProcess( ) in your Fortran code.

Commands run with the SYSTEM routine are run in a separate shell. Defaults set with the SYSTEM function, such as current working directory or environment variables, do not affect the environment the calling program runs in.

The command line character limit for the SYSTEM function is the same limit that your operating system command interpreter accepts.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) I, errnum
I = SYSTEM("dir > file.lst")
If (I .eq. -1) then
  errnum = ierrno( )
  print *, 'Error ', errnum
end if
END
```

See Also

- S
- SYSTEMQQ

SYSTEM_CLOCK

Intrinsic Subroutine: *Returns integer data from a real-time clock. SYSTEM_CLOCK returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. The number is returned with no bias. To get the elapsed time, you must call SYSTEM_CLOCK twice, and subtract the starting time value from the ending time value.*

Syntax

```
CALL SYSTEM_CLOCK ([count] [, count_rate] [, count_max])
```

count	(Output; optional) Must be scalar and of type integer. It is set to a value based on the current value of the processor clock. The value is increased by one for each clock count until the value *count_max* is reached, and is reset to zero at the next count. (*count* lies in the range 0 to *count_max*.)
count_rate	(Output; optional) Must be scalar and of type integer. It is set to the number of processor clock counts per second. If the type is INTEGER(2), *count_rate* is 1000. If the type is INTEGER(4), *count_rate* is 10000. If the type is INTEGER(8), *count_rate* is 1000000.
count_max	(Output; optional) Must be scalar and of type integer. It is set to the maximum value that *count* can have, HUGE(0).

All arguments used must have the same integer kind parameter. If the type is INTEGER(1), *count*, *count_rate*, and *count_max* are all zero, indicating that there is no clock available to Intel Fortran with an 8-bit range.

Example

Consider the following:

```
integer(2) :: ic2, crate2, cmax2
integer(4) :: ic4, crate4, cmax4
call system_clock(count=ic2, count_rate=crate2, count_max=cmax2)
call system_clock(count=ic4, count_rate=crate4, count_max=cmax4)
print *, ic2, crate2, cmax2
print *, ic4, crate4, cmax4
end
```

This program was run on Thursday Dec 11, 1997 at 14:23:55 EST and produced the following output:

```
13880   1000  32767
1129498807       10000  2147483647
```

See Also

- S
- DATE_AND_TIME
- HUGE
- GETTIM

SYSTEMQQ

Portability Function: *Executes a system command by passing a command string to the operating system's command interpreter.*

Module

USE IFPORT

Syntax

`result = SYSTEMQQ (commandline)`

commandline	(Input) Character*(*). Command to be passed to the operating system.

Results

The result type is LOGICAL(4). The result is .TRUE. if successful; otherwise, .FALSE..

The SYSTEMQQ function lets you pass operating-system commands as well as programs. SYSTEMQQ refers to the COMSPEC and PATH environment variables that locate the command interpreter file (usually named COMMAND.COM).

On Windows* systems, the calling process waits until the command terminates. To insure compatibility and consistent behavior, an image can be invoked directly by using the Windows API CreateProcess( ) in your Fortran code.

If the function fails, call GETLASTERRORQQ to determine the reason. One of the following errors can be returned:

- ERR$2BIG - The argument list exceeds 128 bytes, or the space required for the environment formation exceeds 32K.
- ERR$NOINT - The command interpreter cannot be found.
- ERR$NOEXEC - The command interpreter file has an invalid format and is not executable.
- ERR$NOMEM - Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

The command line character limit for the SYSTEMQQ function is the same limit that your operating system command interpreter accepts.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT

LOGICAL(4) result

result = SYSTEMQQ('copy c:\bin\fmath.dat &

                        c:\dat\fmath2.dat')
```

See Also

- S
- SYSTEM

T to Z

TAN

Elemental Intrinsic Function (Generic):
Produces the tangent of x.

Syntax

```
result = TAN (x)
```

x	(Input) Must be of type real or complex. If *x* is of type real, it must be in radians and is treated as modulo 2 * pi. If *x* is of type complex, its real part is regarded as a value in radians.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TAN	REAL(4)	REAL(4)
DTAN	REAL(8)	REAL(8)
QTAN	REAL(16)	REAL(16)
CTAN[1]	COMPLEX(4)	COMPLEX(4)
CDTAN[2]	COMPLEX(8)	COMPLEX(8)
CQTAN	COMPLEX(16)	COMPLEX(16)

[1]The setting of compiler options specifying real size can affect CTAN.

[2]This function can also be specified as ZTAN.

Example

TAN (2.0) has the value -2.185040.

TAN (0.8) has the value 1.029639.

TAND

Elemental Intrinsic Function (Generic):
Produces the tangent of x.

Syntax

```
result = TAND (x)
```

x	(Input) Must be of type real. It must be in degrees and is treated as modulo 360.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TAND	REAL(4)	REAL(4)
DTAND	REAL(8)	REAL(8)
QTAND	REAL(16)	REAL(16)

Example

TAND (2.0) has the value 3.4920771E-02.

TAND (0.8) has the value 1.3963542E-02.

TANH

Elemental Intrinsic Function (Generic):
Produces a hyperbolic tangent.

Syntax

```
result = TANH (x)
```

x	(Input) Must be of type real.

Results

The result type is the same as *x*.

Specific Name	Argument Type	Result Type
TANH	REAL(4)	REAL(4)
DTANH	REAL(8)	REAL(8)
QTANH	REAL(16)	REAL(16)

Example

TANH (2.0) has the value 0.9640276.

TANH (0.8) has the value 0.6640368.

TARGET

Statement and Attribute: *Specifies that an object can become the target of a pointer (it can be pointed to).*

Syntax

The TARGET attribute can be specified in a type declaration statement or a TARGET statement, and takes one of the following forms:

Type Declaration Statement:

type,[*att-ls*,] TARGET [, *att-ls*] :: *object*[(*a-spec*)][, *object*[(*a-spec*)]]...

Statement:

TARGET [::]*object*[(*a-spec*)][, *object*[(*a-spec*)]] ...

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
object	Is the name of the object. The object must not be declared with the PARAMETER attribute.
a-spec	(Optional) Is an array specification.

Description

A pointer is associated with a target by pointer assignment or by an ALLOCATE statement.

If an object does not have the TARGET attribute or has not been allocated (using an ALLOCATE statement), no part of it can be accessed by a pointer.

Example

The following example shows type declaration statements specifying the TARGET attribute:

```
TYPE(SYSTEM), TARGET :: FIRST
REAL, DIMENSION(20, 20), TARGET :: C, D
```

The following is an example of a TARGET statement:

```
TARGET :: C(50, 50), D
```

The following fragment is from the program POINTER2.F90 in the `<install-dir>/samples` subdirectory:

```
 ! An example of pointer assignment.
     REAL, POINTER :: arrow1 (:)
     REAL, POINTER :: arrow2 (:)
     REAL, ALLOCATABLE, TARGET :: bullseye (:)
     ALLOCATE (bullseye (7))
     bullseye = 1.
     bullseye (1:7:2) = 10.
     WRITE (*,'(/1x,a,7f8.0)') 'target ',bullseye
     arrow1 => bullseye
     WRITE (*,'(/1x,a,7f8.0)') 'pointer',arrow1
     . . .
```

See Also

- T to Z
- ALLOCATE
- ASSOCIATED
- POINTER
- Pointer Assignments
- Pointer Association
- Type Declarations
- Compatible attributes

TASK

OpenMP* Fortran Compiler Directive: *Defines a task region.*

Syntax

```
c$OMP TASK [clause[[,] clause] ... ]
   block
c$OMP END TASK
```

c — Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

clause — Is one of the following:

- DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
- FIRSTPRIVATE (list)
- IF (*scalar_logical_expression*)

  Specifies that the enclosed code section is to be executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. If this clause is not used, the region is executed as if an IF(.TRUE.) clause were specified.

  If the *scalar_logical_expression* evaluates to .FALSE., the encountering thread must suspend the current task region and begin execution of the generated task immediately. The suspended task region will not be resumed until the generated task is completed.

  This clause is evaluated by the master thread before any data scope attributes take effect.

  Only a single IF clause can appear in the directive.
- PRIVATE (list)
- SHARED (list)
- UNTIED

Specifies the task is never tied to the thread that started its execution. Any thread in the team can resume the task region after a suspension. For example, during runtime, the compiler can start the execution of a given task on thread A, break execution and later resume it on thread B.

block Is a structured block (section) of statements or constructs. You cannot branch into or out of the block (the parallel region).

The TASK and END TASK directive pair must appear in the same routine in the executable section of the code.

The END TASK directive denotes the end of the task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.

A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

The task construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at its point of completion.

Note that when storage is shared by an explicit task region, you must add proper synchronization to ensure that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

Example

The following example calculates a Fibonacci number. The Fibonacci sequence is 1,1,2,3,5,8,13, etc., where the current number is the sum of the previous two numbers. If a call to function fib is encountered by a single thread in a parallel region, a nested task region will be spawned to carry out the computation in parallel.

```
RECURSIVE INTEGER FUNCTION fib(n)
INTEGER n, i, j
IF ( n .LT. 2) THEN
  fib = n
ELSE
  !$OMP TASK SHARED(i)
     i = fib( n-1 )
  !$OMP END TASK
  !$OMP TASK SHARED(j)
     j = fib( n-2 )
  !$OMP END TASK
  !$OMP TASKWAIT      ! wait for the sub-tasks to

        !   complete before summing
     fib = i+j
END IF
END FUNCTION
```

The following example generates a large number of tasks in one thread and executes them with the threads in the parallel team. While generating these tasks, if the implementation reaches the limit generating unassigned tasks, the generating loop may be suspended and the thread used to execute unassigned tasks. When the number of unassigned tasks is sufficiently low, the thread resumes execution of the task generating loop.

```
real*8 item(10000000)
integer i
!$omp parallel
!$omp single  ! loop iteration variable i is private
    do i=1,10000000
!$omp task
! i is firstprivate, item is shared
    call process(item(i))
!$omp end task
    end do
!$omp end single
!$omp end parallel
end
```

The following example modifies the previous one to use an untied task to generate the unassigned tasks. If the implementation reaches the limit generating unassigned tasks and the generating loop is suspended, any other thread that becomes available can resume the task generation loop.

```
real*8 item(10000000)
!$omp parallel
!$omp single
!$omp task untied
! loop iteration variable i is private
    do i=1,10000000
!$omp task  ! i is firstprivate, item is shared
    call process(item(i))
!$omp end task
    end do
!$omp end task
!$omp end single
!$omp end parallel
```

See Also

- T to Z
- OpenMP Fortran Compiler Directives
- OpenMP* Fortran Routines
- PARALLEL DO
- TASKWAIT
- SHARED Clause

TASKWAIT

OpenMP* Fortran Compiler Directive: *Specifies a wait on the completion of child tasks generated since the beginning of the current task.*

Syntax

```
c$OMP TASKWAIT
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

The TASKWAIT region includes an implicit task scheduling point in the current task region. The current task region is suspended at the task scheduling point until execution of all its child tasks generated before the TASKWAIT region are completed.

See Also

- T to Z
- OpenMP Fortran Compiler Directives
- TASK directive

THREADPRIVATE

OpenMP* Fortran Compiler Directive: *Specifies named common blocks to be private (local) to a thread; they are global within the thread.*

Syntax

```
c$OMP THREADPRIVATE (/ cb/ [,/ cb/]...)
```

c	Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).
cb	Is the name of the common block you want made private to a thread. Only named common blocks can be made thread private. Note that the slashes (/) are required.

Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads.

During serial portions and MASTER sections of the program, accesses are to the master thread copy of the common block. On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified in the PARALLEL directive.

When a common block (which is initialized using DATA statements) appears in a THREADPRIVATE directive, each thread copy is initialized once prior to its first use. For subsequent parallel regions, data in THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

A THREADPRIVATE common block or its constituent variables can appear only in a COPYIN clause. They are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

Example

In the following example, the common blocks BLK1 and FIELDS are specified as thread private:

```
      COMMON /BLK/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
c$OMP THREADPRIVATE(/BLK/,/FIELDS/)
c$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/,ZFIELD)
```

See Also

- T to Z
- OpenMP Fortran Compiler Directives

TIME Intrinsic Procedure

Intrinsic Subroutine (Generic): *Returns the current time as set within the system. TIME can be used as an intrinsic subroutine or as a portability function or subroutine. It is an intrinsic procedure unless you specify USE IFPORT. Intrinsic subroutines cannot be passed as actual arguments.*

Syntax

CALL TIME (*buf*)

buf	(Output) Is a variable, array, or array element of any data type, or a character substring. It must contain at least eight bytes ofstorage.

The date is returned as a 8-byte ASCII character string taking the form hh:mm:ss, where:

hh	is the 2-digit hour
mm	is the 2-digit minute
ss	is the 2-digit second

If *buf* is of numeric type and smaller than 8 bytes, data corruption can occur.

If *buf* is of character type, its associated length is passed to the subroutine. If *buf* is smaller than 8 bytes, the subroutine truncates the date to fit in the specified length. If an array of type character is passed, the subroutine stores the date in the first array element, using the element length, not the length of the entire array.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS LIB

Example

```
CHARACTER*1 HOUR(8)
...
CALL TIME (HOUR)
```

The length of the first array element in CHARACTER array HOUR is passed to the TIME subroutine. The subroutine then truncates the time to fit into the 1-character element, producing an incorrect result.

See Also

- T to Z
- DATE_AND_TIME
- TIME portability routine

TIME Portability Routine

Portability Function or Subroutine: *The function returns the system time, in seconds, since 00:00:00 Greenwich mean time, January 1, 1970. TIME can be used as an intrinsic subroutine or as a portability function or subroutine. It is an intrinsic procedure unless you specify USE IFPORT.*

Module

USE IFPORT

Syntax

Function Syntax:

```
result = TIME( )
```

Subroutine Syntax:

```
CALL TIME (timestr)
```

timestr	(Output) Character*(*). Is the current time, based on a 24-hour clock, in the form hh:mm:ss, where hh, mm, and ss are two-digit representations of the current hour, minutes past the hour, and seconds past the minute, respectively.

Results

The result type is INTEGER(4). The result value is the number of seconds that have elapsed since 00:00:00 Greenwich mean time, January 1, 1970.

The subroutine fills a parameter with the current time as a string in the format hh:mm:ss.

The value returned by this routine can be used as input to other portability date and time functions.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) int_time
character*8 char_time
int_time = TIME( )
call TIME(char_time)
print *, 'Integer: ', int_time, 'time: ', char_time
END
```

See Also

- T to Z
- DATE_AND_TIME
- TIME intrinsic procedure

TIMEF

Portability Function: *Returns the number of seconds since the first time it is called, or zero.*

Module

USE IFPORT

Syntax

```
result = TIMEF( )
```

Results

The result type is REAL(4). The result value is the number of seconds that have elapsed since the first time TIMEF was called.

The first time it is called, TIMEF returns 0.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER i, j
REAL(8) elapsed_time
elapsed_time = TIMEF( )
DO i = 1, 100000
   j = j + 1
END DO
elapsed_time = TIMEF( )
PRINT *, elapsed_time
END
```

See Also

- T to Z
- Date and Time Procedures

TINY

Inquiry Intrinsic Function (Generic): *Returns the smallest number in the model representing the same type and kind parameters as the argument.*

Syntax

```
result = TINY (x)
```

x	(Input) Must be of type real; it can be scalar or array valued.

Results

The result is a scalar with the same type and kind parameters as *x*. The result has the value $b^{e_{min-1}}$. Parameters *b* and e_{min} are defined in Model for Real Data.

Example

If X is of type REAL(4), TINY (X) has the value 2^{-126}.

The following shows another example:

```
REAL(8) r, result

r = 487923.3D0

result = TINY(r)    ! returns 2.225073858507201E-308
```

See Also

- T to Z
- HUGE
- Data Representation Models

TRACEBACKQQ

Run-Time Subroutine: *Provides traceback information. Uses the Intel® Fortran run-time library traceback facility to generate a stack trace showing the program call stack as it appeared at the time of the call to TRACEBACKQQ( ).*

Module

USE IFCORE

Syntax

CALL TRACEBACKQQ ([*string*] [,*user_exit_code*] [,*status*] [,*eptr*])

string (Input; optional) CHARACTER*(*). A message string to precede the traceback output. It is recommended that the string be no more than 80 characters (one line) since that length appears better on output. However, this limit is not a restriction and it is not enforced. The string is output exactly as specified; no formatting or interpretation is done.
If this argument is omitted, no header message string is produced.

user_exit_code (Input; optional) INTEGER(4). An exit code. Two values are predefined:

- A value of -1 causes the run-time system to return execution to the caller after producing traceback.
- A value of zero (the default) causes the application to abort execution.

Any other specified value causes the application to abort execution and return the specified value to the operating system.

status (Input; optional) INTEGER(4). A status value. If specified, the run-time system returns the status value to the caller indicating that the traceback process was successful. The default is not to return status.
Note that a returned status value is only an indication that the "attempt" to trace the call stack was completed successfully, not that it produced a useful result.
You can include the file `iosdef.for`in your program to obtain symbolic definitions for the possible return values. A return value of FOR$IOS_SUCCESS (0) indicates success.

eptr (Input; optional) Integer pointer. It is required if calling from a user-specified exception filter. If omitted, the default in null.
To trace the stack after an exception has occurred, the runtime support needs access to the exception information supplied to the filter by the operating system.
The *eptr* argument is a pointer to a T_EXCEPTION_POINTERS structure, which is defined in `ifcore.f90`. This argument is optional and is usually omitted. On Windows systems,

T_EXCEPTION_POINTERS is returned by the Windows* API GetExceptionInformation( ), which is usually passed to a C try/except filter function.

The TRACEBACKQQ routine provides a standard way for an application to initiate a stack trace. It can be used to report application detected errors, debugging, and so forth. It uses the stack trace support in the Intel Fortran run-time library, and produces the same output that the run-time library produces for unhandled errors and exceptions.

The error message string normally included by the run-time system is replaced with the user-supplied message text, or omitted if no string is specified. Traceback output is directed to the target destination appropriate for the application type, just as it is when traceback is initiated internally by the run-time system.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

The following example generates a traceback report with no leading header message, from wherever the call site is, and aborts execution:

```
USE IFCORE

CALL TRACEBACKQQ( )
```

The following example generates a traceback report with the user-supplied string as the header, and aborts execution:

```
USE IFCORE

CALL TRACEBACKQQ("My application message string")
```

The following example generates a traceback report with the user-supplied string as the header, and aborts execution, returning a status code of 123 to the operating system:

```
USE IFCORE

CALL TRACEBACKQQ(STRING="Bad value for TEMP",USER_EXIT_CODE=123)
```

Consider the following:

```
...
USE IFCORE
INTEGER(4) RTN_STS
INCLUDE 'IOSDEF.FOR'
...
CALL TRACEBACKQQ(USER_EXIT_CODE=-1,STATUS=RTN_STS)
IF (RTN_STS .EQ. FOR$IOS_SUCCESS) THEN
  PRINT *,'TRACEBACK WAS SUCCESSFUL'
END IF
...
```

This example generates a traceback report with no header string, and returns to the caller to continue execution of the application. If the traceback process succeeds, a status will be returned in variable RTN_STS.

For more examples, including one showing an integer pointer, see *Building Applications*: Obtaining Traceback Information with TRACEBACKQQ.

See Also

- T to Z
- GETEXCEPTIONPTRSQQ

Building Applications: Using Traceback Information

Building Applications: Run-Time Message Display and Format

Building Applications: Obtaining Traceback Information with TRACEBACKQQ

TRAILZ

Elemental Intrinsic Function (Generic):
Returns the number of trailing zero bits in an integer.

Syntax

```
result = TRAILZ (i)
```

i	(Input) Must be of type integer or logical.

Results

The result type is the same as *i*. The result value is the number of trailing zeros in the binary representation of the integer *i*.

The model for the interpretation of an integer value as a sequence of bits is shown in Model for Bit Data.

Example

Consider the following:

```
INTEGER*8 J, TWO

PARAMETER (TWO=2)

DO J= -1, 40

  TYPE *, TRAILZ(TWO**J)  ! Prints 64, then 0 up to

ENDDO                     !   40 (trailing zeros)

END
```

TRANSFER

Transformational Intrinsic Function (Generic): *Converts the bit pattern of the first argument according to the type and kind parameters of the second argument.*

Syntax

`result = TRANSFER (`*source*`,`*mold*`[,`*size*`])`

source	(Input) Must be a scalar or array (of any data type).
mold	(Input) Must be a scalar or array (of any data type). It provides the type characteristics (not a value) for the result.
size	(Input; optional) Must be scalar and of type integer. It provides the number of elements for the output result.

Results

The result has the same type and type parameters as *mold*.

If *mold* is a scalar and *size* is omitted, the result is a scalar.

If *mold* is an array and *size* is omitted, the result is a rank-one array. Its size is the smallest that is possible to hold all of *source*.

If *size* is present, the result is a rank-one array of size *size*.

If the physical representation of the result is larger than *source*, the result contains *source*'s bit pattern in its right-most bits; the left-most bits of the result are undefined.

If the physical representation of the result is smaller than *source*, the result contains the right-most bits of *source*'s bit pattern.

Example

TRANSFER (1082130432, 0.0) has the value 4.0 (on processors that represent the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000 0000 0000 0000 0000).

TRANSFER ((/2.2, 3.3, 4.4/), ((0.0, 0.0))) results in a scalar whose value is (2.2, 3.3).

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/)) results in a complex rank-one array of length 2. Its first element is (2.2,3.3) and its second element has a real part with the value 4.4 and an undefined imaginary part.

TRANSFER ((/2.2, 3.3, 4.4/), (/(0.0, 0.0)/), 1) results in a complex rank-one array having one element with the value (2.2, 3.3).

The following shows another example:

```
COMPLEX CVECTOR(2), CX(1)
!  The next statement sets CVECTOR to
!  [ 1.1 + 2.2i, 3.3 + 0.0i ]
CVECTOR = TRANSFER((/1.1, 2.2, 3.3, 0.0/), &
                   (/(0.0, 0.0)/))
!  The next statement sets CX to [ 1.1 + 2.2i ]
CX = TRANSFER((/1.1, 2.2, 3.3/) , (/(0.0, 0.0)/), &
              SIZE= 1)
WRITE(*,*) CVECTOR
WRITE(*,*) CX
END
```

TRANSPOSE

Transformational Intrinsic Function (Generic):
Transposes an array of rank two.

Syntax

```
result = TRANSPOSE (matrix)
```

matrix (Input) Must be a rank-two array. It may be of any data type.

Results

The result is a rank-two array with the same type and kind parameters as *matrix*. Its shape is (n, m), where (m, n) is the shape of *matrix*. For example, if the shape of *matrix* is (4,6), the shape of the result is (6,4).

Element (i, j) of the result has the value *matrix*(j, i), where *i* is in the range 1 to n, and *j* is in the range 1 to m.

Example

B is the array

```
[ 2  3  4 ]
[ 5  6  7 ]
[ 8  9  1 ].
```

TRANSPOSE (B) has the value

```
[ 2  5  8 ]
[ 3  6  9 ]
[ 4  7  1 ].
```

The following shows another example:

```
INTEGER array(2, 3), result(3, 2)
array = RESHAPE((/1, 2, 3, 4, 5, 6/), (/2, 3/))
! array is  1  3  5
!           2  4  6
result = TRANSPOSE(array)
! result is 1  2
!           3  4
!           5  6
END
```

See Also

- T to Z
- RESHAPE
- PRODUCT

TRIM

Transformational Intrinsic Function (Generic):
Returns the argument with trailing blanks removed.

Syntax

`result = TRIM (`*`string`*`)`

`string`	(Input) Must be a scalar of type character.

Results

The result is of type character with the same kind parameter as *`string`*. Its length is the length of *`string`* minus the number of trailing blanks in *`string`*.

The value of the result is the same as *`string`*, except any trailing blanks are removed. If *`string`* contains only blank characters, the result has zero length.

Example

TRIM (' NAME ') has the value ' NAME'.

TRIM (' C D ') has the value ' C D'.

The following shows another example:

```
! next line prints 28
WRITE(*, *) LEN("I have blanks behind me       ")
! the next line prints 23
WRITE(*,*) LEN(TRIM("I have blanks behind me       "))
END
```

See Also

- T to Z
- LEN_TRIM

TTYNAM

Portability Subroutine: *Specifies a terminal device name.*

Module

USE IFPORT

Syntax

CALL TTYNAM (*string*,*lunit*)

string	(Output) Character*(*). Name of the terminal device. If the Fortran logical unit is not connected to a terminal, it returns a string filled with blanks.
lunit	(Input) INTEGER(4). A Fortran logical unit number.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

TYPE Statement (Derived Types)

Statement: *Declares a variable to be a derived type. It specifies the name of the user-defined type and the types of its components.*

Syntax

TYPE [[,*type-attr-spec-list*] ::] *name*

component-definition

[*component-definition*]. . .

END TYPE [*name*]

type-attr-spec-list Is *access-spec* or BIND (C).

access-spec Is the PUBLIC or PRIVATE keyword. The keyword can only be specified if the derived-type definition is in the specification part of a module.

name Is the name of the derived data type. It must not be the same as the name of any intrinsic type, or the same as the name of a derived type that can be accessed from a module.

component-definition Is one or more type declaration statements defining the component of derived type.
The first component definition can be preceded by an optional PRIVATE or SEQUENCE statement. (Only one PRIVATE or SEQUENCE statement can appear in a given derived-type definition.)
If SEQUENCE is present, all derived types specified in component definitions must be sequence types.
A *component definition* takes the following form:
type[[, *attr*] ::] *component*[(*a-spec*)] [**char-len*] [*init-ex*]

type Is a type specifier. It can be an intrinsic type or a previously defined derived type. (If the POINTER attribute follows this specifier, the type can also be any accessible derived type, including the type being defined.)

attr Is an optional POINTER attribute for a pointer component, or an optional DIMENSION or ALLOCATABLE attribute for an array component. You cannot specify both the ALLOCATABLE and POINTER attribute. If DIMENSION is specified, it can be followed by an array specification.
Each attribute can only appear once in a given *component-definition*.

component Is the name of the component being defined.

a-spec	Is an optional array specification, enclosed in parentheses. If POINTER or ALLOCATABLE is specified, the array is deferred shape; otherwise, it is explicit shape. In an explicit-shape specification, each bound must be a constant scalar integer expression. If the array bounds are not specified here, they must be specified following the DIMENSION attribute.
char-len	Is an optional scalar integer literal constant; it must be preceded by an asterisk (*). This parameter can only be specified if the component is of type CHARACTER.
init-ex	Is an initialization expression, or for pointer components, => NULL( ). This is a Fortran 95 feature. If *init-ex* is specified, a double colon must appear in the component definition. The equals assignment symbol (=) can only be specified for nonpointer components. The initialization expression is evaluated in the scoping unit of the type definition.

Description

If a name is specified following the END TYPE statement, it must be the same name that follows TYPE in the derived type statement.

A derived type can be defined only once in a scoping unit. If the same derived-type name appears in a derived-type definition in another scoping unit, it is treated independently.

A component name has the scope of the derived-type definition only. Therefore, the same name can be used in another derived-type definition in the same scoping unit.

Two data entities have the same type if they are both declared to be of the same derived type (the derived-type definition can be accessed from a module or a host scoping unit).

If the entities are in different scoping units, they can also have the same derived type if they are declared with reference to different derived-type definitions, and if both derived-type definitions have all of the following:

- The same name

- A SEQUENCE statement (they both have sequence type)
- Components that agree in name, order, and attributes; components cannot be private

If BIND (C) is specified, the following rules apply:

- The derived type cannot be a SEQUENCE type.
- The derived type must have type parameters.
- Each component of the derived type must be a nonpointer, nonallocatable data component with interoperable type and type parameters.

Example

```
!   DERIVED.F90
!   Define a derived-type structure,
!   type variables, and assign values
    TYPE member
      INTEGER age
      CHARACTER (LEN = 20) name
    END TYPE member
    TYPE (member) :: george
    TYPE (member) :: ernie
    george     = member( 33, 'George Brown' )
    ernie%age  = 56
    ernie%name = 'Ernie Brown'
    WRITE (*,*) george
    WRITE (*,*) ernie
    END
```

The following shows another example of a derived type:

```
TYPE mem_name
  SEQUENCE
  CHARACTER (LEN = 20) lastn
  CHARACTER (LEN = 20) firstn
  CHARACTER (len = 3) cos   ! this works because COS is a component name
END TYPE mem_name
TYPE member
    TYPE (mem_name) :: name
    SEQUENCE
    INTEGER age
    CHARACTER (LEN = 20) specialty
END TYPE member
```

In the following example, `a` and `b` are both variable arrays of derived type `pair`:

```
 TYPE (pair)
   INTEGER i, j
 END TYPE
 TYPE (pair), DIMENSION (2, 2) :: a, b(3)
```

The following example shows how you can use derived-type objects as components of other derived-type objects:

```
TYPE employee_name
  CHARACTER(25) last_name
  CHARACTER(15) first_name
END TYPE
TYPE employee_addr
  CHARACTER(20) street_name
  INTEGER(2) street_number
  INTEGER(2) apt_number
  CHARACTER(20) city
  CHARACTER(2) state
  INTEGER(4) zip
END TYPE
```

Objects of these derived types can then be used within a third derived-type specification, such as:

```
TYPE employee_data
  TYPE (employee_name) :: name
  TYPE (employee_addr) :: addr
  INTEGER(4) telephone
  INTEGER(2) date_of_birth
  INTEGER(2) date_of_hire
  INTEGER(2) social_security(3)
  LOGICAL(2) married
  INTEGER(2) dependents
END TYPE
```

See Also

- C to D
- E to F

- T to Z
- DIMENSION
- MAP...END MAP
- PRIVATE
- PUBLIC
- RECORD
- SEQUENCE
- STRUCTURE...END STRUCTURE
- Derived Data Types
- Default Initialization
- Structure Components
- Structure Constructors

Building Applications: Handling User-Defined Types

Type Declarations

Statement: *Explicitly specifies the properties of data objects or functions.*

Syntax

A type declaration statement has the general form:

```
type[ [, att] ... :: ] v[/c-list/][, v[/c-list/]] ...
```

type Is one of the following data type specifiers:
BYTE
INTEGER[([KIND=]k)]
REAL[([KIND=]k)]
DOUBLE PRECISION
COMPLEX[([KIND=]k)]
DOUBLE COMPLEX
CHARACTER[([KIND=]k)]
LOGICAL[([KIND=]k)]
TYPE (derived-type-name)
In the optional kind selector "([KIND=]k)", *k* is the kind parameter. It must be an acceptable kind parameter for that data type. If the kind selector is not present, entities declared are of default type.

Kind parameters for intrinsic numeric and logical data types can also be specified using the *n format, where *n* is the length (in bytes) of the entity; for example, INTEGER*4.
See each data type for further information on that type.

att Is one of the following attribute specifiers:

ALLOCATABLE	INTENT	PROTECTED
ASYNCHRONOUS	INTRINSIC	PUBLIC[1]
AUTOMATIC	OPTIONAL	SAVE
BIND	PARAMETER	STATIC
DIMENSION	POINTER	TARGET
EXTERNAL	PRIVATE[1]	VOLATILE

[1]These are access specifiers.

You can also declare any attribute separately as a statement.

att Is the name of a data object or function. It can optionally be followed by:

- An array specification, if the object is an array.

  In a function declaration, an array must be a deferred-shape array if it has the POINTER attribute; otherwise, it must be an explicit-shape array.
- A character length, if the object is of type character.
- An initialization expression preceded by an = or, for pointer objects, => NULL( ).

A function name must be the name of an intrinsic function, external function, function dummy procedure, or statement function.

c-list Is a list of constants, as in a DATA statement. If *v* has the PARAMETER attribute, the *c-list* cannot be present.

The *c-list* cannot specify more than one value unless it initializes an array. When initializing an array, the *c-list* must contain a value for every element in the array.

Description

Type declaration statements must precede all executable statements.

In most cases, a type declaration statement overrides (or confirms) the implicit type of an entity. However, a variable that appears in a DATA statement and is typed implicitly can appear in a subsequent type declaration only if that declaration confirms the implicit typing.

The double colon separator (::) is required only if the declaration contains an attribute specifier or initialization; otherwise it is optional.

If *att* appears, *c-list* cannot be specified; for example:

```
INTEGER I /2/              ! Valid

INTEGER, SAVE :: I /2/     ! Invalid
```

The same attribute must not appear more than once in a given type declaration statement, and an entity cannot be given the same attribute more than once in a scoping unit.

If the PARAMETER attribute is specified, the declaration must contain an initialization expression.

If => NULL( ) is specified for a pointer, its initial association status is disassociated.

A variable (or variable subobject) can only be initialized once in an executable program.

The INTENT, VALUE, and OPTIONAL attributes can be specified only for dummy arguments.

The VALUE attribute must not be specified for a dummy procedure.

If a declaration contains an initialization expression, but no PARAMETER attribute is specified, the object is a variable whose value is initially defined. The object becomes defined with the value determined from the initialization expression according to the rules of intrinsic assignment.

The presence of initialization implies that the name of the object is saved, except for objects in named common blocks or objects with the PARAMETER attribute.

The following objects cannot be initialized in a type declaration statement:

- A dummy argument
- A function result
- An object in a named common block (unless the type declaration is in a block data program unit)

- An object in blank common
- An allocatable array
- An external name
- An intrinsic name
- An automatic object
- An object that has the AUTOMATIC attribute

An object can have more than one attribute. The following table lists the compatible attributes:

Table 929: Compatible Attributes

Attribute	Compatible with:
ALLOCATABLE	AUTOMATIC, ASYNCHRONOUS, DIMENSION [1], PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
ASYNCHRONOUS	ALLOCATABLE, AUTOMATIC, BIND, DIMENSION, INTENT, OPTIONAL, POINTER, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VALUE, VOLATILE
AUTOMATIC	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PROTECTED, TARGET, VOLATILE
BIND	ASYNCHRONOUS, AUTOMATIC, DIMENSION, EXTERNAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
DIMENSION	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, INTENT, OPTIONAL, PARAMETER, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET, VOLATILE
EXTERNAL	BIND, OPTIONAL, PRIVATE, PUBLIC
INTENT	ASYNCHRONOUS, DIMENSION, OPTIONAL, TARGET, VOLATILE

Attribute	Compatible with:
INTRINSIC	PRIVATE, PUBLIC
OPTIONAL	ASYNCHRONOUS, DIMENSION, EXTERNAL, INTENT, POINTER, TARGET, VALUE, VOLATILE
PARAMETER	DIMENSION, PRIVATE, PUBLIC
POINTER	ASYNCHRONOUS, AUTOMATIC, DIMENSION [1], OPTIONAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, VOLATILE
PRIVATE	ASYNCHRONOUS, ALLOCATABLE, BIND, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, PROTECTED, SAVE, STATIC, TARGET, VOLATILE
PROTECTED	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PRIVATE, PUBLIC, SAVE, TARGET, VOLATILE
PUBLIC	ASYNCHRONOUS, ALLOCATABLE, BIND, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, PROTECTED, SAVE, STATIC, TARGET, VOLATILE
SAVE	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PRIVATE, PROTECTED, PUBLIC, STATIC, TARGET, VOLATILE
STATIC	ALLOCATABLE, ASYNCHRONOUS, BIND, DIMENSION, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, TARGET, VOLATILE
TARGET	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, DIMENSION, INTENT, OPTIONAL, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, VALUE, VOLATILE

3630

Attribute	Compatible with:
VALUE	ASYNCHRONOUS, INTENT (IN only), OPTIONAL, TARGET
VOLATILE	ALLOCATABLE, ASYNCHRONOUS, AUTOMATIC, BIND, DIMENSION, INTENT, OPTIONAL, POINTER, PRIVATE, PROTECTED, PUBLIC, SAVE, STATIC, TARGET

[1]With deferred shape

Example

The following show valid type declaration statements:

```
DOUBLE PRECISION B(6)

INTEGER(KIND=2) I

REAL(KIND=4) X, Y

REAL(4) X, Y

LOGICAL, DIMENSION(10,10) :: ARRAY_A, ARRAY_B

INTEGER, PARAMETER :: SMALLEST = SELECTED_REAL_KIND(6, 70)

REAL(KIND (0.0)) M

COMPLEX(KIND=8) :: D

TYPE(EMPLOYEE) :: MANAGER

REAL, INTRINSIC :: COS

CHARACTER(15) PROMPT

CHARACTER*12, SAVE :: HELLO_MSG

INTEGER COUNT, MATRIX(4,4), SUM

LOGICAL*2 SWITCH

REAL :: X = 2.0

TYPE (NUM), POINTER :: FIRST => NULL()
```

The following shows more examples:

```
REAL a (10)
LOGICAL, DIMENSION (5, 5) :: mask1, mask2
COMPLEX :: cube_root = (-0.5, 0.867)
INTEGER, PARAMETER :: short = SELECTED_INT_KIND (4)
REAL (KIND (0.0D0)) a1
REAL (KIND = 2) b
COMPLEX (KIND = KIND (0.0D0)) :: c
INTEGER (short) k ! Range at least -9999 to 9999
TYPE (member) :: george
```

See Also

- T to Z
- CHARACTER
- COMPLEX
- DOUBLE COMPLEX
- DOUBLE PRECISION
- INTEGER
- LOGICAL
- REAL
- IMPLICIT
- RECORD
- STRUCTURE
- TYPE
- Type Declaration Statements

DEFINE and UNDEFINE

General Compiler Directives: *DEFINE creates a symbolic variable whose existence or value can be tested during conditional compilation. UNDEFINE removes a defined symbol.*

Syntax

```
cDEC$ DEFINE name[ = val]
```

*c*DEC$ UNDEFINE *name*

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
name	Is the name of the variable.
val	INTEGER(4). The value assigned to *name*.

DEFINE creates and UNDEFINE removes symbols for use with the IF (or IF DEFINED) compiler directive. Symbols defined with DEFINE directive are local to the directive. They cannot be declared in the Fortran program.

Because Fortran programs cannot access the named variables, the names can duplicate Fortran keywords, intrinsic functions, or user-defined names without conflict.

To test whether a symbol has been defined, use the IF DEFINED (*name*) directive. You can assign an integer value to a defined symbol. To test the assigned value of *name*, use the IF directive. IF test expressions can contain most logical and arithmetic operators.

Attempting to undefine a symbol that has not been defined produces a compiler warning.

The DEFINE and UNDEFINE directives can appear anywhere in a program, enabling and disabling symbol definitions.

Example

```
!DEC$ DEFINE testflag
!DEC$ IF DEFINED (testflag)
   write (*,*) 'Compiling first line'
!DEC$ ELSE
   write (*,*) 'Compiling second line'
!DEC$ ENDIF
!DEC$ UNDEFINE testflag
```

See Also

- C to D
- T to Z
- IF Directive Construct
- General Compiler Directives
- D compiler option

Building Applications: Compiler Directives Related to Options

Building Applications: Using Predefined Preprocessor Symbols

UNION...END UNION

Statements: *Define a data area that can be shared intermittently during program execution by one or more fields or groups of fields. A union declaration must be within a structure declaration.*

Syntax

Each unique field or group of fields is defined by a separate map declaration.

```
UNION
   map-declaration
   map-declaration
   [map-declaration]
   . . .
   [map-declaration]
END UNION
```

map-declaration Takes the following form:

```
MAP
   field-declaration
   [field-declaration]
   . . .
   [field-declaration]
END MAP
```

field-declaration Is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a data field (having a data type) within a union. It can be of any intrinsic or derived type.

As with normal Fortran type declarations, data can be initialized in field declaration statements in union declarations. However, if fields within multiple map declarations in a single union are initialized, the data declarations are initialized in the order in which the statements appear. As a result, only the final initialization takes effect and all of the preceding initializations are overwritten.

The size of the shared area established for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the fields declared within it.

Manipulating data by using union declarations is similar to using EQUIVALENCE statements. The difference is that data entities specified within EQUIVALENCE statements are concurrently associated with a common storage location and the data residing there; with union declarations you can use one discrete storage location to alternately contain a variety of fields (arrays or variables).

With union declarations, only one map declaration within a union declaration can be associated at any point in time with the storage location that they share. Whenever a field within another map declaration in the same union declaration is referenced in your program, the fields in the prior map declaration become undefined and are succeeded by the fields in the map declaration containing the newly referenced field.

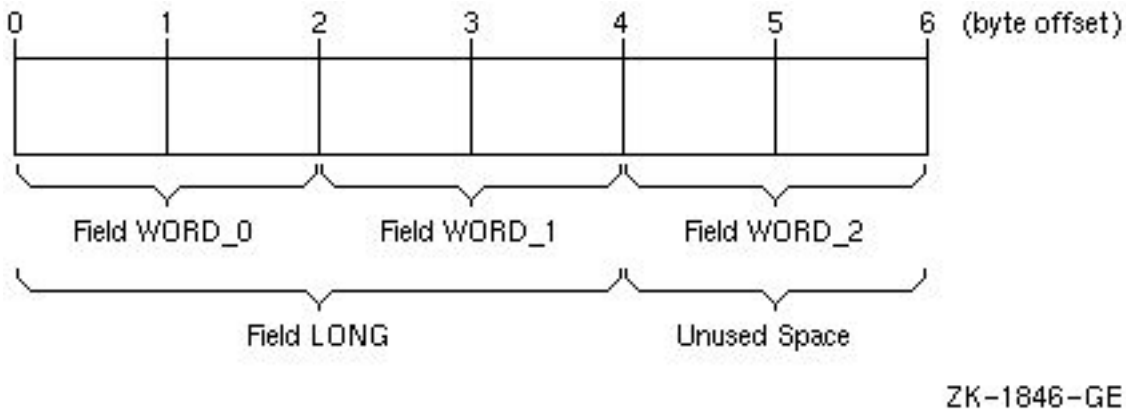
Example

In the following example, the structure WORDS_LONG is defined. This structure contains a union declaration defining two map fields. The first map field consists of three INTEGER*2 variables (WORD_0, WORD_1, and WORD_2), and the second, an INTEGER*4 variable, LONG:

```
 STRUCTURE /WORDS_LONG/
   UNION
     MAP
       INTEGER*2  WORD_0, WORD_1, WORD_2
     END MAP
     MAP
       INTEGER*4  LONG
     END MAP
   END UNION
 END STRUCTURE
```

The length of any record with the structure WORDS_LONG is 6 bytes. The following figure shows the memory mapping of any record with the structure WORDS_LONG:

Figure 105: Memory Map of Structure WORDS_LONG



In the following example, note how the first 40 characters in the string2 array are overlayed on 4-byte integers, while the remaining 20 are overlayed on 2-byte integers:

```
  UNION
     MAP
       CHARACTER*20 string1, CHARACTER*10 string2(6)
     END MAP
     MAP
       INTEGER*2 number(10), INTEGER*4 var(10), INTEGER*2
 +      datum(10)
     END MAP
  END UNION
```

See Also

- E to F
- T to Z
- STRUCTURE...END STRUCTURE
- Record Structures

UNLINK

Portability Function: *Deletes the file given by path.*

Module

USE IFPORT

Syntax

`result = UNLINK (name)`

name	(Input) Character*(*). Path of the file to delete. The path can use forward (/) or backward (\) slashes as path separators and can contain drive letters.

Results

The result type is INTEGER(4). The result is zero if successful; otherwise, an error code. Errors can be one of the following:

- ENOENT: The specified file could not be found.
- EACCES: The specified file is read-only.

You must have adequate permission to delete the specified file.

On Windows systems, you will get the EACCES error if the file has been opened by any process.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
INTEGER(4) ISTATUS
CHARACTER*20 dirname
READ *, dirname
ISTATUS = UNLINK (dirname)
IF (ISTATUS) then
   print *, 'Error ', ISTATUS
END IF
END
```

See Also

- T to Z
- SYSTEM
- DELDIRQQ

UNPACK

Transformational Intrinsic Function (Generic): *Takes elements from a rank-one array and unpacks them into another (possibly larger) array under the control of a mask.*

Syntax

`result = UNPACK (`*`vector`*`,`*`mask`*`,`*`field`*`)`

vector	(Input) Must be a rank-one array. It may be of any data type. Its size must be at least *t*, where *t* is the number of true elements in *mask*.
mask	(Input) Must be a logical array. It determines where elements of *vector* are placed when they are unpacked.
field	(Input) Must be of the same type and type parameters as *vector* and conformable with *mask*. Elements in *field* are inserted into the result array when the corresponding *mask* element has the value false.

Results

The result is an array with the same shape as *mask*, and the same type and type parameters as *vector*.

Elements in the result array are filled in array element order. If element i of *mask* is true, the corresponding element of the result is filled by the next element in *vector*. Otherwise, it is filled by *field* (if *field* is scalar) or the ith element of *field* (if *field* is an array).

Example

N is the array

```
[ 0  0  1 ]
[ 1  0  1 ]
[ 1  0  0 ],
```

P is the array (2, 3, 4, 5), and Q is the array

```
[ T  F  F ]
[ F  T  F ]
[ T  T  F ].
```

UNPACK (P, MASK=Q, FIELD=N) produces the result

```
[ 2  0  1 ]
[ 1  4  1 ]
[ 3  5  0 ].
```

UNPACK (P, MASK=Q, FIELD=1) produces the result

```
[ 2  1  1 ]
[ 1  4  1 ]
[ 3  5  1 ].
```

The following shows another example:

```
LOGICAL mask (2, 3)
INTEGER vector(3) /1, 2, 3/, AR1(2, 3)
mask = RESHAPE((/.TRUE.,.FALSE.,.FALSE.,.TRUE.,&
                .TRUE.,.FALSE./), (/2, 3/))
! vector = [1 2 3] and mask =  T F T
!                              F T F
AR1 = UNPACK(vector, mask, 8)  ! returns  1 8 3
                               !          8 2 8
END
```

See Also

- T to Z
- PACK
- RESHAPE
- SHAPE

UNPACKTIMEQQ

Portability Subroutine: *Unpacks a packed time and date value into its component parts.*

Module

USE IFPORT

Syntax

CALL UNPACKTIMEQQ (*timedate*,*iyr*,*imon*,*iday*,*ihr*,*imin*,*isec*)

timedate	(Input) INTEGER(4). Packed time and date information.
iyr	(Output) INTEGER(2). Year (*xxxx*AD).
imon	(Output) INTEGER(2). Month (1 - 12).
iday	(Output) INTEGER(2). Day (1 - 31).
ihr	(Output) INTEGER(2). Hour (0 - 23).
imin	(Output) INTEGER(2). Minute (0 - 59).
isec	(Output) INTEGER(2). Second (0 - 59).

GETFILEINFOQQ returns time and date in a packed format. You can use UNPACKTIMEQQ to unpack these values. Use PACKTIMEQQ to repack times for passing to SETFILETIMEQQ. Packed times can be compared using relational operators.

Compatibility

CONSOLE STANDARD GRAPHICS QUICKWIN GRAPHICS WINDOWS DLL LIB

Example

```
USE IFPORT
CHARACTER(80)    file
TYPE (FILE$INFO) info
INTEGER(4) handle, result
INTEGER(2) iyr, imon, iday, ihr, imin, isec
file = 'd:\f90ps\bin\t???.*'
handle = FILE$FIRST
result = GETFILEINFOQQ(file, info, handle)
CALL UNPACKTIMEQQ(info%lastwrite, iyr, imon,&
                  iday, ihr, imin, isec)
WRITE(*,*) iyr, imon, iday
WRITE(*,*) ihr, imin, isec
END
```

See Also

- T to Z
- PACKTIMEQQ
- GETFILEINFOQQ

UNREGISTERMOUSEEVENT (W*32, W*64)

QuickWin Function: *Removes the callback routine registered for a specified window by an earlier call to REGISTERMOUSEEVENT.*

Module

USE IFQWIN

Syntax

```
result = UNREGISTERMOUSEEVENT (unit,mouseevents)
```

unit (Input) INTEGER(4) on IA-32 architecture; INTEGER(8) on Intel® 64 architecture and IA-64 architecture. Unit number of the window whose callback routine on mouse events is to be unregistered.

mouseevents (Input) INTEGER(4). One or more mouse events handled by the callback routine to be unregistered. Symbolic constants (defined in `IFQWIN.F90`) for the possible mouse events are:

- MOUSE$LBUTTONDOWN - Left mouse button down
- MOUSE$LBUTTONUP - Left mouse button up
- MOUSE$LBUTTONDBLCLK - Left mouse button double-click
- MOUSE$RBUTTONDOWN - Right mouse button down
- MOUSE$RBUTTONUP - Right mouse button up
- MOUSE$RBUTTONDBLCLK - Right mouse button double-click
- MOUSE$MOVE - Mouse moved

Results

The result type is INTEGER(4). The result is zero or a positive integer if successful; otherwise, a negative integer that can be one of the following:

- MOUSE$BADUNIT - The unit specified is not open, or is not associated with a QuickWin window.
- MOUSE$BADEVENT - The event specified is not supported.

Once you call UNREGISTERMOUSEEVENT, QuickWin no longer calls the callback routine specified earlier for the window when mouse events occur. Calling UNREGISTERMOUSEEVENT when no callback routine is registered for the window has no effect.

Compatibility

QUICKWIN GRAPHICS LIB

See Also

- T to Z
- REGISTERMOUSEEVENT

- WAITONMOUSEEVENT

Building Applications: Using QuickWin Overview

UNROLL and NOUNROLL

General Compiler Directive: *Tells the compiler's optimizer how many times to unroll a DO loop or disables the unrolling of a DO loop. These directives can only be applied to iterative DO loops.*

Syntax

```
cDEC$ UNROLL [(n)] -or- cDEC$ UNROLL [=n]

cDEC$ NOUNROLL
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is an integer constant. The range of *n* is 0 through 255.

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop.

The UNROLL directive overrides any setting of loop unrolling from the command line.

To use these directives, compiler option O2 or O3 must be set.

Example

```
cDEC$ UNROLL
  do i =1, m
    b(i) = a(i) + 1
    d(i) = c(i) + 1
  enddo
```

See Also

- M to N
- T to Z
- General Compiler Directives
- Rules for General Directives that Affect DO Loops
- O compiler option

UNROLL_AND_JAM and NOUNROLL_AND_JAM

General Compiler Directive: *Enables or disables loop unrolling and jamming. These directives can only be applied to iterative DO loops.*

Syntax

*c*DEC$ UNROLL_AND_JAM [(*n*)] -or- *c*DEC$ UNROLL_AND_JAM [=*n*]

*c*DEC$ NOUNROLL_AND_JAM

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
n	Is an integer constant. The range of *n* is 0 through 255.

Description

If *n* is specified, the optimizer unrolls the loop *n* times. If *n* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop. The loops are partially unrolled and then the resulting loops are fused ("jammed") back together.

The UNROLL_AND_JAM directive overrides any setting of loop unrolling from the command line. It takes precedence over all other unrolling directives, including NOUNROLL.

To use these directives, compiler option O2 or O3 must be set.

Example

```
cDEC$ UNROLL_AND_JAM = 4
  do i =1, m
    b(i) = a(c(i)) + 1
  enddo
```

See Also

- T to Z
- General Compiler Directives
- Rules for General Directives that Affect DO Loops
- O compiler option

USE

Statement: *Gives a program unit accessibility to public entities in a module.*

Syntax

```
USE [[, mod-nature] ::] name [, rename-list]...
USE [[, mod-nature] ::] name, ONLY : [, only-list]
```

mod-nature — Is INTRINSIC or NON_INTRINSIC. If INTRINSIC is used, *name* must be the name of an intrinsic module. If NON_INTRINSIC is used, *name* must be the name of an nonintrinsic module. If *mod-nature* is not specified, *name* must be the name of an intrinsic or nonintrinsic module. If both are provided, the nonintrinsic module is used. It is an error to specify a user module and an intrinsic module of the same name in the same program unit (see Examples).

name — Is the name of the module.

rename-list — Is one or more items having the following form:
local-name=> *mod-name*

- *local-name* — Is the name of the entity in the program unit using the module or is "OPERATOR (*op-name*)", where *op-name* is the name of a defined operator in the program unit using the module.
- *mod-name* — Is the name of a public entity in the module or is "OPERATOR (*op-name*)", where *op-name* is the name of a public entity in the module.

only-list — Is the name of a public entity in the module or a generic identifier (a generic name, a defined operator specified as "OPERATOR (*op-name*)",or defined assignment).
An entity in the *only-list* can also take the form:
[*local-name* =>] *mod-name*

Description

If the USE statement is specified without the ONLY option, the program unit has access to all public entities in the named module.

If the USE statement is specified with the ONLY option, the program unit has access to only those entities following the option.

If more than one USE statement for a given module appears in a scoping unit, the following rules apply:

- If one USE statement does not have the ONLY option, all public entities in the module are accessible, and any *rename-list*s and *only-list*s are interpreted as a single, concatenated *rename-list*.
- If all the USE statements have ONLY options, all the *only-list*s are interpreted as a single, concatenated *only-list*. Only those entities named in one or more of the *only-list*s are accessible.

If two or more generic interfaces that are accessible in a scoping unit have the same name, the same operator, or are both assignments, they are interpreted as a single generic interface. Otherwise, multiple accessible entities can have the same name only if no reference to the name is made in the scoping unit.

The local names of entities made accessible by a USE statement must not be respecified with any attribute other than PUBLIC or PRIVATE. The local names can appear in namelist group lists, but not in a COMMON or EQUIVALENCE statement.

The following shows examples of the USE statement:

```
MODULE MOD_A
  INTEGER :: B, C
  REAL E(25,5), D(100)
END MODULE MOD_A
...
SUBROUTINE SUB_Y
  USE MOD_A, DX => D, EX => E   ! Array D has been renamed DX and array E
  ...                           ! has been renamed EX. Scalar variables B
END SUBROUTINE SUB_Y            ! and C are also available to this subrou-
...                             ! tine (using their module names).
SUBROUTINE SUB_Z
  USE MOD_A, ONLY: B, C         ! Only scalar variables B and C are
  ...                           !   available to this subroutine
END SUBROUTINE SUB_Z
...
```

You must not specify a user module and an intrinsic module of the same name in the same program unit. For example, if you specify a user module named ISO_FORTRAN_ENV, then it is illegal to specify the following in the same program unit:

```
USE :: ISO_FORTRAN_ENV
USE, INTRINSIC :: ISO_FORTRAN_ENV
```

The following example shows a module containing common blocks:

```
MODULE COLORS
  COMMON /BLOCKA/ C, D(15)
  COMMON /BLOCKB/ E, F
  ...
END MODULE COLORS
...
FUNCTION HUE(A, B)
  USE COLORS
  ...
END FUNCTION HUE
```

The USE statement makes all of the variables in the common blocks in module COLORS available to the function HUE.

To provide data abstraction, a user-defined data type and operations to be performed on values of this type can be packaged together in a module. The following example shows such a module:

```
MODULE CALCULATION
  TYPE ITEM
    REAL :: X, Y
  END TYPE ITEM
  INTERFACE OPERATOR (+)
    MODULE PROCEDURE ITEM_CALC
  END INTERFACE
CONTAINS
  FUNCTION ITEM_CALC (A1, A2)
    TYPE(ITEM) A1, A2, ITEM_CALC
    ...
  END FUNCTION ITEM_CALC
  ...
END MODULE CALCULATION
PROGRAM TOTALS
USE CALCULATION
TYPE(ITEM) X, Y, Z
  ...
  X = Y + Z
  ...
END
```

The USE statement allows program TOTALS access to both the type ITEM and the extended intrinsic operator + to perform calculations.

The following shows another example:

```
! Module containing original type declarations
MODULE geometry
type square
   real side
   integer border
end type
type circle
   real radius
   integer border
end type
END MODULE
! Program renames module types for local use.
PROGRAM test
USE GEOMETRY,LSQUARE=>SQUARE,LCIRCLE=>CIRCLE
! Now use these types in declarations
type (LSQUARE) s1,s2
type (LCIRCLE) c1,c2,c3
```

The following shows a defined operator in a USE statement:

```
USE mymod, OPERATOR(.localop.) => OPERATOR(.moduleop.)
```

See Also

- T to Z
- Program Units and Procedures
- USE overview

%VAL

Built-in Function: *Changes the form of an actual argument. Passes the argument as an immediate value.*

Syntax

```
%VAL (a)
```

a	(Input) An expression, record name, procedure name, array, character array section, or array element.

Description

The argument is passed as follows:

- On IA-32 architecture, as a 32-bit immediate value. If the argument is integer (or logical) and shorter than 32 bits, it is sign-extended to a 32-bit value. For complex data types, %VAL passes two 32-bit arguments.
- On Intel® 64 architecture and IA-64 architecture, as a 64-bit immediate value. If the argument is integer (or logical) and shorter than 64 bits, it is sign-extended to a 64-bit value. For complex data types, %VAL passes two 64-bit arguments.

You must specify %VAL in the actual argument list of a CALL statement or function reference. You cannot use it in any other context.

The following tables list the Intel Fortran defaults for argument passing, and the allowed uses of %VAL:

Table 930: Expressions

Actual Argument Data Type	Default	%VAL
Logical	REF	Yes[1]
Integer	REF	Yes[1]
REAL(4)	REF	Yes
REAL(8)	REF	Yes[2]
REAL(16)	REF	No

Actual Argument Data Type	Default	%VAL
COMPLEX(4)	REF	Yes
COMPLEX(8)	REF	Yes
COMPLEX(16)	REF	No
Character	See table note [3]	No
Hollerith	REF	No
Aggregate[4]	REF	No
Derived	REF	No

Table 931: Array Name

Actual Argument Data Type	Default	%VAL
Numeric	REF	No
Character	See table note [3]	No
Aggregate[4]	REF	No
Derived	REF	No

Table 932: Procedure Name

Actual Argument Data Type	Default	%VAL
Numeric	REF	No
Character	See table note [3]	No

The %VAL and %REF functions override related cDEC$ ATTRIBUTE settings.

Example

```
CALL SUB(2, %VAL(2))
```

Constant 2 is passed by reference. The second constant 2 is passed by immediate value.

[1] If a logical or integer value occupies less than 64 bits of storage on Intel® 64 architecture and IA-64 architecture, or 32 bits of storage on IA-32 architecture, it is converted to the correct size by sign extension. Use the ZEXT intrinsic function if zero extension is desired.

[2] i64 only

[3] A character argument is passed by address and hidden length.

[4] In Intel Fortran record structures

See Also

- T to Z
- CALL
- %REF
- %LOC

VALUE

Statement and Attribute: *Specifies a type of argument association for a dummy argument.*

Syntax

The VALUE attribute can be specified in a type declaration statement or a VALUE statement, and takes one of the following forms:

Type Declaration Statement:

```
type, [att-ls, ] VALUE [att-ls,] :: arg [, arg] ...
```

Statement:

```
VALUE [::] arg [, arg]...
```

type	Is a data type specifier.
att-ls	Is an optional list of attribute specifiers.
arg	Is the name of a dummy argument.

Description

The VALUE attribute can be used in INTERFACE body or in a procedure. It can only be specified for dummy arguments. It cannot be specified for a dummy procedure.

When this attribute is specified, the effect is as if the actual argument is assigned to a temporary, and the temporary is the argument associated with the dummy argument. The actual mechanism by which this happens is determined by the processor.

When the VALUE attribute is used in a type declaration statement, any length type parameter values must be omitted or they must be specified by initialization expressions.

If the VALUE attribute is specified, you cannot specify a PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, or INTENT (INOUT or OUT) attribute in the same scoping unit.

Example

The following example shows how the VALUE attribute can be applied in a type declaration statement.

```
j = 3
call sub (j)
write (*,*) j ! Writes 3
contains
subroutine sub (i)
integer, value :: I
i = 4
write (*,*) i ! Writes 4
end subroutine sub
end
```

See Also

- T to Z
- Type Declarations
- Compatible attributes

VECTOR ALIGNED and VECTOR UNALIGNED

General Compiler Directive: *Specifies that all data is aligned or no data is aligned in a DO loop.*

Syntax

```
cDEC$ VECTOR ALIGNED
cDEC$ VECTOR UNALIGNED
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

Description

These directives override efficiency heuristics in the optimizer. The qualifiers UNALIGNED and ALIGNED instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

CAUTION. The directives VECTOR ALIGNED and VECTOR UNALIGNED should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.

Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

See Also

- T to Z
- T to Z
- Rules for General Directives that Affect DO Loops

VECTOR ALWAYS and NOVECTOR

General Compiler Directive: *Enables or disables vectorization of a DO loop.*

Syntax

```
cDEC$ VECTOR ALWAYS
```

```
cDEC$ NOVECTOR
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

The VECTOR ALWAYS and NOVECTOR directives override the default behavior of the compiler. The VECTOR ALWAYS directive also overrides efficiency heuristics of the vectorizer, but it only works if the loop can actually be vectorized. You should use the IVDEP directive to ignore assumed dependences.

CAUTION. The directive VECTOR ALWAYS should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.

Example

The compiler normally does not vectorize DO loops that have a large number of non-unit stride references (compared to the number of unit stride references).

In the following example, vectorization would be disabled by default, but the directive overrides this behavior:

```
!DEC$ VECTOR ALWAYS
  do i = 1, 100, 2
    ! two references with stride 2 follow
    a(i) = b(i)
  enddo
```

There may be cases where you want to explicitly avoid vectorization of a loop; for example, if vectorization would result in a performance regression rather than an improvement. In these cases, you can use the NOVECTOR directive to disable vectorization of the loop.

In the following example, vectorization would be performed by default, but the directive overrides this behavior:

```
!DEC$ NOVECTOR
  do i = 1, 100
    a(i) = b(i) + c(i)
  enddo
```

See Also

- M to N
- T to Z
- Rules for General Directives that Affect DO Loops

VECTOR TEMPORAL and VECTOR NONTEMPORAL (i32, i64em)

General Compiler Directive: *Controls how the "stores" of register contents to storage are performed (streaming versus non-streaming). These directives are only available on IA-32 architecture and Intel® 64 architecture.*

Syntax

```
cDEC$ VECTOR TEMPORAL
cDEC$ VECTOR NONTEMPORAL [(var[, var]...)]
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
val	Is an optional memory reference in the form of a variable name.

Description

VECTOR NONTEMPORAL directs the compiler to use non-temporal (that is, streaming) stores. VECTOR TEMPORAL directs the compiler to use temporal (that is, non-streaming) stores.

By default, the compiler automatically determines whether a streaming store should be used for each variable.

Streaming stores may cause significant performance improvements over non-streaming stores for large numbers on certain processors. However, the misuse of streaming stores can significantly degrade performance.

For more information on this directive, including an example, see *Optimizing Applications*: Vectorization Support.

See Also

- T to Z
- T to Z
- Rules for General Directives that Affect DO Loops

VECTOR TEMPORAL and VECTOR NONTEMPORAL (i32, i64em)

General Compiler Directive: *Controls how the "stores" of register contents to storage are performed (streaming versus non-streaming). These directives are only available on IA-32 architecture and Intel® 64 architecture.*

Syntax

```
cDEC$ VECTOR TEMPORAL
cDEC$ VECTOR NONTEMPORAL [(var[, var]...)]
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)
val	Is an optional memory reference in the form of a variable name.

Description

VECTOR NONTEMPORAL directs the compiler to use non-temporal (that is, streaming) stores. VECTOR TEMPORAL directs the compiler to use temporal (that is, non-streaming) stores.

By default, the compiler automatically determines whether a streaming store should be used for each variable.

Streaming stores may cause significant performance improvements over non-streaming stores for large numbers on certain processors. However, the misuse of streaming stores can significantly degrade performance.

For more information on this directive, including an example, see *Optimizing Applications*: Vectorization Support.

See Also

- T to Z
- T to Z
- Rules for General Directives that Affect DO Loops

VECTOR ALIGNED and VECTOR UNALIGNED

General Compiler Directive: *Specifies that all data is aligned or no data is aligned in a DO loop.*

Syntax

```
cDEC$ VECTOR ALIGNED
cDEC$ VECTOR UNALIGNED
```

c	Is one of the following: C (or c), !, or *. (See Syntax Rules for Compiler Directives.)

Description

These directives override efficiency heuristics in the optimizer. The qualifiers UNALIGNED and ALIGNED instruct the compiler to use, respectively, unaligned and aligned data movement instructions for all array references. This disables all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

CAUTION. The directives VECTOR ALIGNED and VECTOR UNALIGNED should be used with care. Overriding the efficiency heuristics of the compiler should only be done if you are absolutely sure the vectorization will improve performance.

Furthermore, instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

See Also

- T to Z
- T to Z
- Rules for General Directives that Affect DO Loops

VERIFY

Elemental Intrinsic Function (Generic): *Verifies that a set of characters contains all the characters in a string by identifying the first character in the string that is not in the set.*

Syntax

```
result = VERIFY (string, set [, back] [, kind])
```

string	(Input) Must be of type character.
set	(Input) Must be of type character with the same kind parameter as *string*.
back	(Input; optional) Must be of type logical.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

If *back* is omitted (or is present with the value false) and *string* has at least one character that is not in *set*, the value of the result is the position of the leftmost character of *string* that is not in *set*.

If *back* is present with the value true and *string* has at least one character that is not in *set*, the value of the result is the position of the rightmost character of *string* that is not in *set*.

If each character of *string* is in *set* or the length of *string* is zero, the value of the result is zero.

The setting of compiler options specifying integer size can affect this function.

Example

VERIFY ('CDDDC', 'C') has the value 2.

VERIFY ('CDDDC', 'C', BACK=.TRUE.) has the value 4.

VERIFY ('CDDDC', 'CD') has the value zero.

The following shows another example:

```
INTEGER(4) position

position = VERIFY ('banana', 'nbc')  ! returns 2

position = VERIFY ('banana', 'nbc', BACK=.TRUE.)

                                     ! returns 6

position = VERIFY ('banana', 'nbca') ! returns 0
```

See Also

- T to Z
- SCAN

VIRTUAL

Statement: *Has the same form and effect as the DIMENSION statement. It is included for compatibility with PDP-11 FORTRAN.*

See Also

- T to Z
- DIMENSION

VOLATILE

Statement and Attribute: *Specifies that the value of an object is entirely unpredictable, based on information local to the current program unit. It prevents objects from being optimized during compilation.*

Syntax

The VOLATILE attribute can be specified in a type declaration statement or a VOLATILE statement, and takes one of the following forms:

Type Declaration Statement:

type, [*att-ls*,] VOLATILE [, *att-ls*] :: *object*[, *object*] ...

Statement:

VOLATILE [::] *object*[, *object*] ...

type Is a data type specifier.

att-ls	Is an optional list of attribute specifiers.
object	Is the name of an object, or the name of a common block enclosed in slashes.

A variable or COMMON block must be declared VOLATILE if it can be read or written in a way that is not visible to the compiler. For example:

- If an operating system feature is used to place a variable in shared memory (so that it can be accessed by other programs), the variable must be declared VOLATILE.
- If a variable is accessed or modified by a routine called by the operating system when an asynchronous event occurs, the variable must be declared VOLATILE.

If an array is declared VOLATILE, each element in the array becomes volatile. If a common block is declared VOLATILE, each variable in the common block becomes volatile.

If an object of derived type is declared VOLATILE, its components become volatile.

If a pointer is declared VOLATILE, the pointer itself becomes volatile.

A VOLATILE statement cannot specify the following:

- A procedure
- A namelist group

Example

The following example shows a type declaration statement specifying the VOLATILE attribute:

```
INTEGER, VOLATILE :: D, E
```

The following example shows a VOLATILE statement:

```
PROGRAM TEST
LOGICAL(KIND=1) IPI(4)
INTEGER(KIND=4) A, B, C, D, E, ILOOK
INTEGER(KIND=4) P1, P2, P3, P4
COMMON /BLK1/A, B, C
VOLATILE /BLK1/, D, E
EQUIVALENCE(ILOOK, IPI)
EQUIVALENCE(A, P1)
EQUIVALENCE(P1, P4)
```

The named common block, BLK1, and the variables D and E are volatile. Variables P1 and P4 become volatile because of the direct equivalence of P1 and the indirect equivalence of P4.

See Also

- T to Z
- Type Declarations
- Compatible attributes

Optimizing Applications: Improving I/O Performance

WAIT

Statement: *Performs a wait operation for a specified pending asynchronous data transfer operation. It takes one of the following forms:*

Syntax

```
WAIT([UNIT=]io-unit [, END=label] [, EOR=label] [, ERR=label] [, ID=id-var]
[, IOSTAT=i-var])

WAIT io-unit
```

io-unit	(Input) Is an external unit specifier.
label	(Input) Is the label of the branch target statement that receives control if an error occurs.
id-var	(Input) Is a scalar integer variable that is the identifier of a pending data transfer operation for the specified unit. If it is specified, a wait operation is performed for that pending operation. If it is omitted, wait operations are performed for all pending data transfers for the specified unit.
i-var	(Output) Is a scalar integer variable that is defined as a positive integer if an error occurs and zero if no error occurs.

A wait operation completes the processing of a pending data transfer operation. Each wait operation completes only a single data transfer operation, although a single statement may perform multiple wait operations.

The WAIT statement specifiers can appear in any order. An I/O unit must be specified, but the UNIT= keyword is optional if the unit specifier is the first item in the I/O control list.

The EOR= specifier only has effect if the pending data transfer operation is a nonadvancing read. The END= specifier only has effect if the pending data transfer operation is a READ.

Example

The following example shows how the WAIT statement can be applied.

```
program test
integer, asynchronous, dimension(100) :: array
open (unit=1,file='asynch.dat',asynchronous='YES',&
  form='unformatted')
write (1) (i,i=1,100)
rewind (1)
read (1,asynchronous='YES') array
wait(1)
write (*,*) array(1:10)
end
```

WAITONMOUSEEVENT (W*32, W*64)

QuickWin Function: *Waits for the specified mouse input from the user.*

Module

USE IFQWIN

Syntax

`result = WAITONMOUSEEVENT (mouseevents,keystate,x,y)`

mouseevents (Input) INTEGER(4). One or more mouse events that must occur before the function returns. Symbolic constants for the possible mouse events are:

- MOUSE$LBUTTONDOWN - Left mouse button down
- MOUSE$LBUTTONUP - Left mouse button up
- MOUSE$LBUTTONDBLCLK - Left mouse button double-click
- MOUSE$RBUTTONDOWN - Right mouse button down
- MOUSE$RBUTTONUP - Right mouse button up
- MOUSE$RBUTTONDBLCLK - Right mouse button double-click

- MOUSE$MOVE - Mouse moved

keystate (Output) INTEGER(4). Bitwise inclusive OR of the state of the mouse during the event. The value returned in *keystate* can be any or all of the following symbolic constants:

- MOUSE$KS_LBUTTON - Left mouse button down during event
- MOUSE$KS_RBUTTON - Right mouse button down during event
- MOUSE$KS_SHIFT - SHIFTkey held down during event
- MOUSE$KS_CONTROL - CTRLkey held down during event

x (Output) INTEGER(4). X position of the mouse when the event occurred.

y (Output) INTEGER(4). Y position of the mouse when the event occurred.

Results

The result type is INTEGER(4). The result is the symbolic constant associated with the mouse event that occurred if successful. If the function fails, it returns the constant MOUSE$BADEVENT, meaning the event specified is not supported.

WAITONMOUSEEVENT does not return until the specified mouse input is received from the user. While waiting for a mouse event to occur, the status bar changes to read "Mouse input pending in XXX", where XXX is the name of the window. When a mouse event occurs, the status bar returns to its previous value.

A mouse event must happen in the window that had focus when WAITONMOUSEEVENT was initially called. Mouse events in other windows will not end the wait. Mouse events in other windows cause callbacks to be called for the other windows, if callbacks were previously registered for those windows.

For every BUTTONDOWN or BUTTONDBLCLK event there is an associated BUTTONUP event. When the user double clicks, four events happen: BUTTONDOWN and BUTTONUP for the first click, and BUTTONDBLCLK and BUTTONUP for the second click. The difference between getting BUTTONDBLCLK and BUTTONDOWN for the second click depends on whether the second click occurs in the double click interval, set in the system's CONTROL PANEL/MOUSE.

Compatibility

QUICKWIN GRAPHICS LIB

Example

```
  USE IFQWIN
  INTEGER(4) mouseevent, keystate, x, y, result
  ...
  mouseevent = MOUSE$RBUTTONDOWN .OR. MOUSE$LBUTTONDOWN
  result = WAITONMOUSEEVENT (mouseevent, keystate, x , y)

!
! Wait until right or left mouse button clicked, then check the keystate
! with the following:
!
  if ((MOUSE$KS_SHIFT .AND. keystate) == MOUSE$KS_SHIFT) then      &
 & write (*,*) 'Shift key was down'
  if ((MOUSE$KS_CONTROL .AND. keystate) == MOUSE$KS_CONTROL) then  &
 & write (*,*) 'Ctrl key was down'
```

See Also

- T to Z

REGISTERMOUSEEVENT

UNREGISTERMOUSEEVENT

Building Applications: Blocking Procedures

WHERE

Statement and Construct: *Lets you use masked array assignment, which performs an array operation on selected elements. This kind of assignment applies a logical test to an array on an element-by-element basis.*

Syntax

Statement:

`WHERE (`*`mask-expr1`*`) assign-stmt`

Construct:

```
[name:]WHERE (mask-expr1)
    [where-body-stmt] ...
[ELSE WHERE(mask-expr2) [name]
    [where-body-stmt] ...]
[ELSE WHERE[name]
    [where-body-stmt] ...]
END WHERE [name]
```

mask-expr1, mask-expr2 Are logical array expressions (called mask expressions).

assign-stmt Is an assignment statement of the form: array variable = array expression.

name Is the name of the WHERE construct.

where-body-stmt Is one of the following:

- An *assign-stmt*

  The assignment can be a defined assignment only if the routine implementing the defined assignment is elemental.

- A WHERE statement or construct

Description

If a construct *name* is specified in a WHERE statement, the same name must appear in the corresponding END WHERE statement. The same construct name can optionally appear in any ELSE WHERE statement in the construct. (ELSE WHERE cannot specify a different name.)

In each assignment statement, the mask expression, the variable being assigned to, and the expression on the right side, must all be conformable. Also, the assignment statement cannot be a defined assignment.

Only the WHERE statement (or the first line of the WHERE construct) can be labeled as a branch target statement.

The following shows an example using a WHERE statement:

```
INTEGER A, B, C
DIMENSION A(5), B(5), C(5)
DATA A /0,1,1,1,0/
DATA B /10,11,12,13,14/
C = -1
WHERE(A .NE. 0) C = B / A
```

The resulting array C contains: -1,11,12,13, and -1.

The assignment statement is only executed for those elements where the mask is true. Think of the mask expression as being evaluated first into a logical array that has the value true for those elements where A is positive. This array of trues and falses is applied to the arrays A, B and C in the assignment statement. The right side is only evaluated for elements for which the mask is true; assignment on the left side is only performed for those elements for which the mask is true. The elements for which the mask is false do not get assigned a value.

In a WHERE construct, the mask expression is evaluated first and only once. Every assignment statement following the WHERE is executed as if it were a WHERE statement with " *mask-expr1*" and every assignment statement following the ELSE WHERE is executed as if it were a WHERE statement with ".NOT. *mask-expr1*". If ELSE WHERE specifies "*mask-expr2*", it is executed as "(.NOT. *mask-expr1*) .AND. *mask-expr2*" during the processing of the ELSE WHERE statement.

You should be careful if the statements have side effects, or modify each other or the mask expression.

The following is an example of the WHERE construct:

```
DIMENSION PRESSURE(1000), TEMP(1000), PRECIPITATION(1000)
WHERE(PRESSURE .GE. 1.0)
  PRESSURE = PRESSURE + 1.0
  TEMP = TEMP - 10.0
ELSEWHERE
  PRECIPITATION = .TRUE.
ENDWHERE
```

The mask is applied to the arguments of functions on the right side of the assignment if they are considered to be elemental functions. Only elemental intrinsics are considered elemental functions. Transformational intrinsics, inquiry intrinsics, and functions or operations defined in the subprogram are considered to be nonelemental functions.

Consider the following example using LOG, an elemental function:

```
WHERE(A .GT. 0)  B = LOG(A)
```

The mask is applied to A, and LOG is executed only for the positive values of A. The result of the LOG is assigned to those elements of B where the mask is true.

Consider the following example using SUM, a nonelemental function:

```
REAL A, B

DIMENSION A(10,10), B(10)

WHERE(B .GT. 0.0)  B = SUM(A, DIM=1)
```

Since SUM is nonelemental, it is evaluated fully for all of A. Then, the assignment only happens for those elements for which the mask evaluated to true.

Consider the following example:

```
REAL A, B, C

DIMENSION A(10,10), B(10), C(10)

WHERE(C .GT. 0.0)  B = SUM(LOG(A), DIM=1)/C
```

Because SUM is nonelemental, all of its arguments are evaluated fully regardless of whether they are elemental or not. In this example, LOG(A) is fully evaluated for all elements in A even though LOG is elemental. Notice that the mask is applied to the result of the SUM and to C to determine the right side. One way of thinking about this is that everything inside the argument list of a nonelemental function does not use the mask, everything outside does.

Example

```
REAL(4) a(20)

. . .

WHERE (a > 0.0)

  a = LOG (a)

     !LOG is invoked only for positive elements

END WHERE
```

See Also

- T to Z
- FORALL
- Arrays

WORKSHARE

OpenMP* Fortran Compiler Directive: *Divides the work of executing a block of statements or constructs into separate units. It also distributes the work of executing the units to threads of the team so each unit is only executed once.*

Syntax

```
c$OMP WORKSHARE
   block
c$OMP END WORKSHARE[ NOWAIT]
```

c — Is one of the following: C (or c), !, or * (see Syntax Rules for Compiler Directives).

block — Is a structured block (section) of statements or constructs. No branching into or out of the block of code is allowed.
The *block* is executed so that each statement is completed before the next statement is started and the evaluation of the right hand side of an assignment is completed before the effects of assigning to the left hand side occur.
The following are additional rules for this argument:

- *block* may contain statements which bind to lexically enclosed PARALLEL constructs. Statements in these PARALLEL constructs are not restricted.
- *block* may contain ATOMIC directives and CRITICAL constructs.
- *block* must only contain array assignment statements, scalar assignment statements, FORALL statements, FORALL constructs, WHERE statements, or WHERE constructs.
- *block* must not contain any user defined function calls unless the function is ELEMENTAL.

If you do not specify the NOWAIT keyword, synchronization is implied following the code.

See Also

- T to Z
- OpenMP Fortran Compiler Directives
- PARALLEL WORKSHARE

WRAPON (W*32, W*64)

Graphics Function: *Controls whether text output is wrapped.*

Module

USE IFQWIN

Syntax

```
result = WRAPON (option)
```

option	(Input) INTEGER(2). Wrap mode. One of the following symbolic constants: • $GWRAPOFF - Truncates lines at right edge of window border. • $GWRAPON - Wraps lines at window border, scrolling if necessary.

Results

The result type is INTEGER(2). The result is the previous value of *option*.

WRAPON controls whether text output with the OUTTEXT function wraps to a new line or is truncated when the text output reaches the edge of the defined text window.

WRAPON does not affect font routines such as OUTGTEXT.

Compatibility

STANDARD GRAPHICS QUICKWIN GRAPHICS LIB

Example

```
USE IFQWIN
INTEGER(2) row, status2
INTEGER(4) status4
TYPE ( rccoord ) curpos
TYPE ( windowconfig ) wc
LOGICAL status
status = GETWINDOWCONFIG( wc )
wc%numtextcols = 80
wc%numxpixels  = -1
wc%numypixels  = -1
wc%numtextrows = -1
wc%numcolors   = -1
wc%fontsize    = -1
wc%title = "This is a test"C
wc%bitsperpixel = -1
status = SETWINDOWCONFIG( wc )
status4= SETBKCOLORRGB(#FF0000 )
CALL CLEARSCREEN( $GCLEARSCREEN )
!  Display wrapped and unwrapped text in text windows.
CALL SETTEXTWINDOW( INT2(1),INT2(1),INT2(5),INT2(25))
CALL SETTEXTPOSITION(INT2(1),INT2(1), curpos )
status2 = WRAPON( $GWRAPON )
status4 = SETTEXTCOLORRGB(#00FF00)
DO i = 1, 5
   CALL OUTTEXT( 'Here text does wrap. ')
END DO
CALL SETTEXTWINDOW(INT2(7),INT2(10),INT2(11),INT2(40))
```

```
CALL SETTEXTPOSITION(INT2(1),INT2(1),curpos)
status2 = WRAPON( $GWRAPOFF )
status4 = SETTEXTCOLORRGB(#008080)
DO row = 1, 5
   CALL SETTEXTPOSITION(INT2(row), INT2(1), curpos )
   CALL OUTTEXT('Here text does not wrap. ')
   CALL OUTTEXT('Here text does not wrap.')
END DO
READ (*,*) ! Wait for ENTER to be pressed
END
```

See Also

- T to Z
- OUTTEXT
- SCROLLTEXTWINDOW
- SETTEXTPOSITION
- SETTEXTWINDOW

WRITE Statement

Statement: *Transfers output data to external sequential, direct-access, or internal records.*

Syntax

Sequential

Formatted:

```
WRITE (eunit, format [, advance] [, asynchronous] [, id] [, pos] [, iostat]
[ ,err]) [io-list]
```

Formatted - List-Directed:

```
WRITE (eunit, * [, asynchronous] [, id] [, pos] [, iostat] [, err]) [io-list]
```

Formatted - Namelist:

```
WRITE (eunit, nml-group [, asynchronous] [, id] [, pos] [, iostat] [, err])
```

Unformatted:

```
WRITE (eunit [, asynchronous] [, id] [, pos] [, iostat] [, err]) [io-list]
```

Direct-Access

Formatted:

```
WRITE (eunit, format, rec [, asynchronous] [, id] [, pos] [, iostat] [, err])
[io-list]
```

Unformatted:

```
WRITE (eunit, rec [, asynchronous] [, id] [, pos] [, iostat] [ , err]) [io-list]
```

Internal

```
WRITE (iunit, format [, iostat] [ , err]) [io-list]
```

eunit	Is an external unit specifier, optionally prefaced by UNIT=. UNIT= is required if *eunit* is not the first specifier in the list.
format	Is a format specifier. It is optionally prefaced by FMT= if *format* is the second specifier in the list and the first specifier indicates a logical or internal unit specifier *without* the optional keyword UNIT=.
advance	Is an advance specifier (ADVANCE=*c-expr*). If the value of *c-expr* is 'YES', the statement uses advancing input; if the value is 'NO', the statement uses nonadvancing input. The default value is 'YES'.
asynchronous	Is an asynchronous specifier (ASYNCHRONOUS=*i-expr*). If the value of *i-expr* is 'YES', the statement uses asynchronous input; if the value is 'NO', the statement uses synchronous input. The default value is 'NO'.
id	Is an id specifier (ID=*id-var*). If ASYNCHRONOUS='YES' is specified and the operation completes successfully, the id specifier becomes defined with an implementation-dependent value that can be specified in a future WAIT or INQUIRE statement to identify the particular data transfer operation. If an error occurs, the id specifier variable becomes undefined.
pos	Is a pos specifier (POS=*p*) that indicates a file position in file storage units in a stream file (ACCESS='STREAM'). It can only be specified on a file opened for stream access. If omitted, the stream I/O occurs starting at the next file position after the current file position.

iostat	Is the name of a variable to contain the completion status of the I/O operation. Optionally prefaced by IOSTAT=.
err	Are branch specifiers if an error (ERR=*label*) condition occurs.
io-list	Is an I/O list: the names of the variables, arrays, array elements, or character substrings from which or to which data will be transferred. Optionally an implied-DO list.
form	Is the nonkeyword form of a format specifier (no FMT=).
*	Is the format specifier indicating list-directed formatting. (It can also be specified as FMT= *.)
nml-group	Is the namelist group specification for namelist I/O. Optionally prefaced by NML=. NML= is required if *nml-group* is not the second I/O specifier. For more information, see Namelist Specifier.
rec	Is the cell number of a record to be accessed directly. Optionally prefaced by REC=.
iunit	Is an internal unit specifier, optionally prefaced by UNIT=. UNIT= is required if *iunit* is not the first specifier in the list. It must be a character variable. It must not be an array section with a vector subscript. If an item in *io-list* is an expression that calls a function, that function must not execute an I/O statement or the EOF intrinsic function on the same external unit as *eunit*.

Example

```
! write to file
open(1,FILE='test.dat')
write (1, '(A20)') namedef
! write with FORMAT statement
WRITE (*, 10) (n, SQRT(FLOAT(n)), FLOAT(n)**(1.0/3.0), n = 1, 100)
10 FORMAT (I5, F8.4, F8.5)
```

The following shows another example:

```
WRITE(6,'("Expected ",F12.6)') 2.0
```

See Also

- T to Z

- I/O Lists
- I/O Control List
- Forms for Sequential WRITE Statements
- Forms for Direct-Access WRITE Statements
- Forms and Rules for Internal WRITE Statements
- READ
- PRINT
- OPEN
- I/O Formatting

XOR

See IEOR.

ZEXT

Elemental Intrinsic Function (Generic):
Extends an argument with zeros. This function is used primarily for bit-oriented operations. It cannot be passed as an actual argument.

Syntax

```
result = ZEXT (x [,kind])
```

x	(Input) Must be of type logical or integer.
kind	(Input; optional) Must be a scalar integer initialization expression.

Results

The result type is integer. If *kind* is present, the kind parameter of the result is that specified by *kind*; otherwise, the kind parameter of the result is that of default integer. If the processor cannot represent the result value in the kind of the result, the result is undefined.

The result value is *x* extended with zeros and treated as an unsigned value.

The storage requirements for integer constants are never less than two bytes. Integer constants within the range of constants that can be represented by a single byte still require two bytes of storage.

The setting of compiler options specifying integer size can affect this function.

Specific Name [1]	Argument Type	Result Type
IZEXT	LOGICAL(1)	INTEGER(2)
	LOGICAL(2)	INTEGER(2)
	INTEGER(1)	INTEGER(2)
	INTEGER(2)	INTEGER(2)
JZEXT	LOGICAL(1)	INTEGER(4)
	LOGICAL(2)	INTEGER(4)
	LOGICAL(4)	INTEGER(4)
	INTEGER(1)	INTEGER(4)
	INTEGER(2)	INTEGER(4)
	INTEGER(4)	INTEGER(4)
KZEXT	LOGICAL(1)	INTEGER(8)
	LOGICAL(2)	INTEGER(8)
	LOGICAL(4)	INTEGER(8)
	LOGICAL(8)	INTEGER(8)
	INTEGER(1)	INTEGER(8)
	INTEGER(2)	INTEGER(8)
	INTEGER(4)	INTEGER(8)
	INTEGER(8)	INTEGER(8)

[1]These specific functions cannot be passed as actual arguments.

Example

Consider the following example:

```
INTEGER(2) W_VAR  /'FFFF'X/

INTEGER(4) L_VAR

L_VAR = ZEXT( W_VAR )
```

This example stores an INTEGER(2) quantity in the low-order 16 bits of an INTEGER(4) quantity, with the resulting value of L_VAR being '0000FFFF'X. If the ZEXT function had not been used, the resulting value would have been 'FFFFFFFF'X, because W_VAR would have been converted to the left-hand operand's data type by sign extension.

Glossary

64

A – B – C – D – E – F – G – H – I – K – L – M – N – O – P – Q – R – S – T – U – V – W – Z

Glossary A

absolute pathname	A directory path specified in fixed relationship to the root directory. On Windows* systems, the first character is a backslash (\). On Linux* and Mac OS* X systems, the first character is a slash (/).
active screen buffer	The screen buffer that is currently displayed in a console's window.
active window	A top-level window of the application with which the user is working. The Windows system identifies the active window by highlighting its title bar and border.
actual argument	A value (a variable, expression, or procedure) passed from a calling program unit to a subprogram (function or subroutine). See also dummy argument.
adjustable array	An explicit-shape array that is a dummy argument to a subprogram. The term is from FORTRAN 77. See also explicit-shape array.
aggregate reference	A reference to a record structure field.
allocatable array	A named array that has the ALLOCATABLE attribute. The array's rank is specified at compile time, but its bounds are determined at run time. Once space has been allocated for this type of array, the array has a shape and can be defined (and redefined) or referenced. It is an error to allocate an allocatable array that is currently allocated.
allocation status	Indicates whether an allocatable array or pointer is allocated. An allocation status is one of: allocated, deallocated, or undefined. An undefined allocation status means an array can no longer be referenced, defined, allocated, or deallocated. See also association status.
alphanumeric	Pertaining to letters and digits.
alternate return	A subroutine argument that permits control to branch immediately to some position other than the statement following the call. The actual argument in an alternate return is the statement label to which control should be transferred. (An alternate return is an obsolescent feature in Fortran 90.)
ANSI	The American National Standards Institute. An organization through which accredited organizations create and maintain voluntary industry standards.
argument	Can be either of the following: • An actual argument--A variable, expression, or procedure passed from a calling program unit to a subprogram. See also actual argument.

	• A dummy argument--A variable whose name appears in the parenthesized list followuing the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. See also dummy argument.
argument association	The relationship (or "matching up") between an actual argument and dummy argument during the execution of a procedure reference.
argument keyword	The name of a dummy (formal) argument. The name is used in a subprogram definition. Argument keywords can be used when the subprogram is invoked to associate dummy arguments with actual arguments, so that the subprogram arguments can appear in any order. Argument keywords are supplied for many of the intrinsic procedures.
array	A set of scalar data that all have the same type and kind type parameters. An array can be referenced by element (using a subscript), by section (using a section subscript list), or as a whole. An array has a rank (up to 7), bounds, size, and a shape. An individual array element is a scalar object. An array section, which is itself an array, is a subset of the entire array. Contrast with scalar. See also bounds, conformable, shape, size, whole array, and zero-sized array.
array constructor	A mechanism used to specify a sequence of scalar values that produce a rank-one array. To construct an array of rank greater than one, you must apply the RESHAPE intrinsic function to the array constructor.
array element	A scalar (individual) item in an array. An array element is identified by the array name followed by one or more subscripts in parentheses, indicating the element's position in the array. For example, B(3) or A(2,5).
array pointer	A pointer to an array. See also array and pointer.
array section	A subobject (or portion) of an array. It consists of the set of array elements or substrings of this set. The set (or section subscript list) is specified by subscripts, subscript triplets, or vector subscripts. If the set does not contain at least one subscript triplet or vector subscript, the reference indicates an array element, not an array.
array specification	A program statement specifying an array name and the number of dimensions the array contains (its rank). An array specification can appear in a DIMENSION or COMMON statement, or in a type declaration statement.
ASCII	The American Standard Code for Information Interchange. A 7-bit character encoding scheme associating an integer from 0 through 127 with 128 characters.

assignment statement	Usually, a statement that assigns (stores) the value of an expression on the right of an equal sign to the storage location of the variable to the left of the equal sign. In the case of Fortran 95/90 pointers, the storage location is assigned, not the pointer itself.
association	The relationship that allows an entity to be referenced by different names in one scoping unit or by the same or different names in more than one scoping unit. The principal kinds of association are argument, host, pointer, storage, and use association. See also argument association, host association, pointer association, storage association, and use association.
association status	Indicates whether or not a pointer is associated with a target. An association status is one of: undefined, associated, or disassociated. An undefined association status means a pointer can no longer be referenced, defined, or deallocated. An undefined pointer can, however, be allocated, nullified, or pointer assigned to a new target. See also allocation status.
assumed-length character argument	A dummy argument that assumes the length attribute of the corresponding actual argument. An asterisk (*) specifies the length of the dummy character argument.
assumed-shape array	A dummy argument array that assumes the shape of its associated actual argument array. The rank of the array is the number of colons (:) specified in parentheses.
assumed-size array	A dummy array whose size (only) is assumed from its associated actual argument. The upper bound of its last dimension is specified by an asterisk (*). All other extents (if any) must be specified.
attribute	A property of a data object that can be specified in a type declaration statement. These properties determine how the data object can be used in a program. Most attributes can be alternatively specified in statements. For example, the DIMENSION statement has the same meaning as the DIMENSION attribute appearing in a type declaration statement.
automatic array	An explicit-shape array that is a local variable in a subprogram. It is not a dummy argument, and has bounds that are nonconstant specification expressions. The bounds (and shape) are determined at entry to the procedure by evaluating the bounds expressions. *See also* automatic object.
automatic object	A local data object that is created upon entry to a subprogram and disappears when the execution of the subprogram is completed. There are two kinds of automatic objects: arrays (of any data type) and objects of type CHARACTER. Automatic objects cannot be saved or initialized.

	An automatic object is not a dummy argument, but is declared with a specification expression that is not a constant expression. The specification expression can be the bounds of the array or the length of the character object.

Glossary B

background process	On Linux* systems, a process for which the command interpreter is not waiting. Its process group differs from that of its controlling terminal, so it is blocked from most terminal access. Contrast with foreground process.
background window	Any window created by a thread other than the foreground thread.
big endian	A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the highest addressed byte. Contrast with little endian.
binary constant	A constant that is a string of binary (base 2) digits (0 or 1) enclosed by apostrophes or quotation marks and preceded by the letter B.
binary operator	An operator that acts on a pair of operands. The exponentiation, multiplication, division, and concatenation operators are binary operators.
bit constant	A constant that is a binary, octal, or hexadecimal number.
bit field	A contiguous group of bits within a binary pattern; they are specified by a starting bit position and length. Some intrinsic functions (for example, IBSET and BTEST) and the intrinsic subroutine MVBITS operate on bit fields.
bitmap	An array of bits that contains data that describes the colors found in a rectangular region on the screen (or the rectangular region found on a page of printer paper).
blank common	A common block (one or more contiguous areas of storage) without a name. Common blocks are defined by a COMMON statement.
block	In general, a group of related items treated as a physical unit. For example, a block can be a group of constructs or statements that perform a task; the task can be executed once, repeatedly, or not at all.
block data program unit	A program unit, containing a BLOCK DATA statement and its associated specification statements, that establishes common blocks and assigns initial values to the variables in named common blocks. In FORTRAN 77, this was called a block data subprogram.

bounds	The range of subscript values for elements of an array. The lower bound is the smallest subscript value in a dimension, and the upper bound is the largest subscript value in that dimension. Array bounds can be positive, zero, or negative. These bounds are specified in an array specification. See also array specification.
breakpoint	A critical point in a program, at which execution is stopped so that you can see if the program variables contain the correct values. Breakpoints are often used to debug programs.
brush	A bitmap that is used to fill the interior of closed shapes, polygons, ellipses, and paths.
brush origin	A coordinate that specifies the location of one of the pixels in a brush's bitmap. The Windows system maps this pixel to the upper left corner of the window that contains the object to be painted. See also bitmap.
built-in procedure	See intrinsic procedure.
byte	A group of 8 contiguous bits (binary digits) starting on an addressable boundary.
byte-order mark	A special Unicode character (0xFEFF) that is placed at the beginning of Unicode text files to indicate that the text is in Unicode format.

Glossary C

carriage-control character	A character in the first position of a printed record that determines the vertical spacing of the output line.
character constant	A constant that is a string of printable ASCII characters enclosed by apostrophes (') or quotation marks (").
character expression	A character constant, variable, function value, or another constant expression, separated by a concatenation operator (//); for example, DAY// ' FIRST'.
character storage unit	The unit of storage for holding a scalar value of default character type (and character length one) that is not a pointer. One character storage unit corresponds to one byte of memory.
character string	A sequence of contiguous characters; a character data value. See also character constant.
character substring	One or more contiguous characters in a character string.
child process	A process initiated by another process (the parent). The child process can operate independently from the parent process. Also, the parent process can suspend or terminate without affecting the child process. See also parent process.
child window	A window that has the WS_CHILD style. A child window always appears within the client area of its parent window. See also parent window.

column-major order	See order of subscript progression.
comment	Text that documents or explains a program. In free source form, a comment begins with an exclamation point (!), unless it appears in a Hollerith or character constant. In fixed and tab source form, a comment begins with a letter C or an asterisk (*) in column 1. A comment can also begin with an exclamation point anywhere in a source line (except in a Hollerith or character constant) or in column 6 of a fixed-format line. The comment extends from the exclamation point to the end of the line. The compiler does not process comments, but shows them in program listings. See also compiler directive.
common block	A physical storage area shared by one or more program units. This storage area is defined by a COMMON statement. If the common block is given a name, it is a named common block; if it is not given a name, it is a blank common. See also blank common and named common block.
compilation unit	The source or files that are compiled together to form a single object file, possibly using interprocedural optimization across source files.
compiler directive	A structured comment that tells the compiler to perform certain tasks when it compiles a source program unit. Compiler directives are usually compiler-specific. (Some Fortran compilers call these directives "metacommands".)
compiler option	An option (or flag) that can be used on the compiler command line to override the default behavior of the Intel® Fortran compiler.
complex constant	A constant that is a pair of real or integer constants representing a complex number; the pair is separated by a comma and enclosed in parentheses. The first constant represents the real part of the number; the second constant represents the imaginary part. The following types of complex constants are available on all systems: COMPLEX(KIND=4), COMPLEX(KIND=8), and COMPLEX(KIND=16).
complex type	A data type that represents the values of complex numbers. The value is expressed as a complex constant. See also data type.
component	Part of a derived-type definition. There must be at least one component (intrinsic or derived type) in every derived-type definition.
concatenate	The combination of two items into one by placing one of the items after the other. In Fortran 95/90, the concatenation operator (//) is used to combine character items. See also character expression.
conformable	Pertains to dimensionality. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.
conformance	See shape conformance.

conservative automatic inlining	The inline expansion of small procedures, with conservative heuristics to limit extra code.
console	An interface that provides input and output to character-mode applications.
constant	A data object whose value does not change during the execution of a program; the value is defined at the time of compilation. A constant can be named (using the PARAMETER attribute or statement) or unnamed. An unnamed constant is called a literal constant. The value of a constant can be numeric or logical, or it can be a character string. Contrast with variable.
constant expression	An expression whose value does not change during program execution.
construct	A series of statements starting with a DO, SELECT CASE, IF, FORALL, or WHERE statement, and ending with the appropriate termination statement.
contiguous	Pertaining to entities that are adjacent (next to one another) without intervening blanks (spaces); for example, contiguous characters or contiguous areas of storage.
control edit descriptor	A format descriptor that directly displays text or affects the conversions performed by subsequent data edit descriptors. Except for the slash descriptor, control edit descriptors are nonrepeatable.
control statement	A statement that alters the normal order of execution by transferring control to another part of a program unit or a subprogram. A control statement can be conditional (such as the IF construct or computed GO TO statement) or unconditional (such as the STOP or GO TO statement).
critical section	An object used to synchronize the threads of a single process. Only one thread at a time can own a critical-section object.

Glossary D

data abstraction	A style of programming in which you define types to represent objects in your program, define a set of operations for objects of each type, and restrict the operations to only this set, making the types abstract. The Fortran 95/90 modules, derived types, and defined operators, support this programming paradigm.
data edit descriptor	A repeatable format descriptor that causes the transfer or conversion of data to or from its internal representation. In FORTRAN 77, this term was called a field descriptor.
data entity	A data object that has a data type. It is the result of the evaluation of an expression, or the result of the execution of a function reference (the function result).

data item	A unit of data (or value) to be processed. Includes constants, variables, arrays, character substrings, or records.
data object	A constant, variable, or subobject (part) of a constant or variable. Its type may be specified implicitly or explicitly.
data type	The properties and internal representation that characterize data and functions. Each intrinsic and user-defined data type has a name, a set of operators, a set of values, and a way to show these values in a program. The basic intrinsic data types are integer, real, complex, logical, and character. The data value of an intrinsic data type depends on the value of the type parameter. See also type parameter.
data type declaration	See type declaration statement.
data type length specifier	The form *n appended to Intel® Fortran-specific data type names. For example, in REAL*4, the *4 is the data type length specifier.
deadlock	A bug where the execution of thread A is blocked indefinitely waiting for thread B to perform some action, while thread B is blocked waiting for thread A. For example, two threads on opposite ends of a named pipe can become deadlocked if each thread waits to read data written by the other thread. A single thread can also deadlock itself. See also thread.
declaration	See specification statement.
decorated name	An internal representation of a procedure name or variable name that contains information about where it is declared; for procedures, the information includes how it is called. Decorated names are mainly of interest in mixed-language programming, when calling Fortran routines from other languages.
default character	The kind for character constants if no kind type parameter is specified. Currently, the only kind type parameter for character constants is CHARACTER(1), the default character kind.
default complex	The kind for complex constants if no kind type parameter is specified. The default complex kind is affected by compiler options specifying double size. If no compiler option is specified, default complex is COMPLEX(4) (COMPLEX*8). See also default real.
default integer	The kind for integer constants if no kind type parameter is specified. The default integer kind is affected by the INTEGER directive, the OPTIONS statement, and by compiler options specifying integer size. If none of these are specified, default integer is INTEGER(4) (INTEGER*4). If a command line option affecting integer size has been specified, the integer has the kind specified, unless it is outside the range of the kind specified by the option. In this case, the kind type of the integer is the smallest integer kind which can hold the integer.

default logical	The kind for logical constants if no kind type parameter is specified. The default logical kind is affected by the INTEGER directive, the OPTIONS statement, and by compiler options specifying integer size. If none of these are specified, default logical is LOGICAL(4) (LOGICAL*4). See also default integer.
default real	The kind for real constants if no kind type parameter is specified. The default real kind is affected by compiler options specifying real size and by the REAL directive. If neither of these is specified, default real is REAL(4) (REAL*4). If a real constant is encountered that is outside the range for the default, an error occurs.
deferred-shape array	An array pointer (an array with the POINTER attribute) or an allocatable array (an array with the ALLOCATABLE attribute). The size in each dimension is determined by pointer assignment or when the array is allocated. The array specification contains a colon (:) for each dimension of the array. No bounds are specified.
definable	A property of variables. A variable is definable if its value can be changed by the appearance of its name or designator on the left of an assignment statement. An example of a variable that is not definable is an allocatable array that has not been allocated.
defined	For a data object, the property of having or being given a valid value.
defined assignment	An assignment statement that is not intrinsic, but is defined by a subroutine and an ASSIGNMENT(=) interface block. See also derived type and interface block.
defined operation	An operation that is not intrinsic, but is defined by a function subprogram containing a generic interface block with the specifier OPERATOR. See also derived type and interface block.
denormalized number	A computational floating-point result smaller than the lowest value in the normal range of a data type (the smallest representable normalized number). You cannot write a constant for a denormalized number.
derived type	A data type that is user-defined and not intrinsic. It requires a type definition to name the type and specify its components (which can be intrinsic or user-defined types). A structure constructor can be used to specify a value of derived type. A component of a structure is referenced using a percent sign (%). Operations on objects of derived types (structures) must be defined by a function with an OPERATOR interface. Assignment for derived types can be defined intrinsically, or be redefined by a subroutine with an ASSIGNMENT(=) interface. Structures can be used as procedure arguments and function results, and can appear in input and output lists. Also called a user-defined type. See also record, the first definition.

designator	A name that references a subobject (part of a data object) that can be defined and referenced separately from other parts of the data object. A designator is the name of the object followed by a selector that selects the subobject. For example, B(3) is a designator for an array element. Also called a subobject designator. See also selector and subobject.
dimension	A range of values for one subscript or index of an array. An array can have from 1 to 7 dimensions. The number of dimensions is the rank of the array.
dimension bounds	See bounds.
direct access	A method for retrieving or storing data in which the data (record) is identified by the record number, or the position of the record in the file. The record is accessed directly (nonsequentially); therefore, all information is equally accessible. Also called random access. Contrast with sequential access.
DLL	See Dynamic Link Library.
double-byte character set (DBCS)	A mapping of characters to their identifying numeric values, in which each value is 2 bytes wide. Double-byte character sets are sometimes used for languages that have more than 256 characters.
double-precision constant	A processor approximation to the value of a real number that occupies 8 bytes of memory and can assume a positive, negative, or zero value. The precision is greater than a constant of real (single-precision) type. For the precise ranges of the double-precision constants, see *Building Applications*: Data Representation Overview. See also denormalized number.
driver program	A program that is the user interface to the language compiler. It accepts command line options and file names and causes one or more language utilities or system programs to process each file.
dummy aliasing	The sharing of memory locations between dummy (formal) arguments and other dummy arguments or COMMON variables that are assigned.
dummy argument	A variable whose name appears in the parenthesized list following the procedure name in a FUNCTION statement, a SUBROUTINE statement, an ENTRY statement, or a statement function statement. A dummy argument takes the value of the corresponding actual argument in the calling program unit (through argument association). Also called a formal argument.
dummy array	A dummy argument that is an array.
dummy pointer	A dummy argument that is a pointer.
dummy procedure	A dummy argument that is specified as a procedure or appears in a procedure reference. The corresponding actual argument must be a procedure.

Dynamic Link Library (DLL)	A separate source module compiled and linked independently of the applications that use it. Applications access the DLL through procedure calls. The code for a DLL is not included in the user's executable image, but the compiler automatically modifies the executable image to point to DLL procedures at run time.

Glossary E

edit descriptor	A descriptor in a format specification. It can be a data edit descriptor, control edit descriptor, or string edit descriptor. See also control edit descriptor, data edit descriptor, and string edit descriptor.
element	See array element.
elemental	Pertains to an intrinsic operation, intrinsic procedure, or assignment statement that is independently applied to either of the following: • The elements of an array • Corresponding elements of a set of conformable arrays and scalars
end-of-file	The condition that exists when all records in a file open for sequential access have been read.
entity	A general term referring to any Fortran 95/90 concept; for example, a constant, a variable, a program unit, a statement label, a common block, a construct, an I/O unit and so forth.
environment variable	A symbolic variable that represents some element of the operating system, such as a path, a filename, or other literal data.
error number	An integer value denoting an I/O error condition, obtained by using the IOSTAT keyword in an I/O statement.
exceptional values	For floating-point numbers, values outside the range of normalized numbers, including denormal (subnormal) numbers, infinity, Not-a-Number (NaN) values, zero, and other architecture-defined numbers.
executable construct	A CASE, DO, IF, WHERE, or FORALL construct.
executable program	A set of program units that include only one main program.
executable statement	A statement that specifies an action to be performed or controls one or more computational instructions.
explicit interface	A procedure interface whose properties are known within the scope of the calling program, and do not have to be assumed. These properties are the names of the procedure and its dummy arguments, the attributes of a procedure (if it is a function), and the attributes and order of the dummy arguments.

	The following have explicit interfaces: • Internal and module procedures (explicit by definition) • Intrinsic procedures • External procedures that have an interface block • External procedures that are defined by the scoping unit and are recursive • Dummy procedures that have an interface block
explicit-shape array	An array whose rank and bounds are specified when the array is declared.
expression	A data reference or a computation formed from operators, operands, and parentheses. The result of an expression is either a scalar value or an array of scalar values.
extent	The size of (number of elements in) one dimension of an array.
external file	A sequence of records that exists in a medium external to the executing program.
external procedure	A procedure that is contained in an external subprogram. External procedures can be used to share information (such as source files, common blocks, and public data in modules) and can be used independently of other procedures and program units. Also called an external routine.
external subprogram	A subroutine or function that is not contained in a main program, module, or another subprogram. A module is not a subprogram.

Glossary F

field	Can be either of the following: • A set of contiguous characters, considered as a single item, in a record or line. • A substructure of a STRUCTURE declaration.
field descriptor	See data edit descriptor.
field separator	The comma (,) or slash (/) that separates edit descriptors in a format specification.
field width	The total number of characters in the field. See also field, the first definition.

file	A collection of logically related records. If the file is in internal storage, it is an internal file; if the file is on an input/output device, it is an external file.
file access	The way records are accessed (and stored) in a file. The Fortran 95/90 file access modes are sequential and direct.
file handle	A unique identifier that the system assigns to a file when the file is opened or created. A file handle is valid until the file is closed.
file organization	The way records in a file are physically arranged on a storage device. Fortran 95/90 files can have sequential or relative organization.
fixed-length record type	A file format in which all the records are the same length.
floating-point environment	A collection of registers that control the behavior of floating-point (FP) machine instructions and indicate the current FP status. The floating-point environment may include rounding mode controls, exception masks, flush-to-zero controls, exception status flags, and other floating-point related features.
focus window	The window to which keyboard input is directed.
foreground process	On Linux* systems, a process for which the command interpreter is waiting. Its process group is the same as that of its controlling terminal, so the process is allowed to read from or write to the terminal. Contrast with background process.
foreground window	The window the user is currently working with. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.
foreign file	An unformatted file that contains data from a foreign platform, such as data from a CRAY*, IBM*, or big endian IEEE* machine.
format	A specific arrangement of data. A FORMAT statement specifies how data is to be read or written.
format specification	The part of a FORMAT statement that specifies explicit data arrangement. It is a list within parentheses that can include edit descriptors and field separators. A character expression can also specify format; the expression must evaluate to a valid format specification.
formatted data	Data written to a file by using formatted I/O statements. Such data contains ASCII representations of binary values.
formatted I/O statement	An I/O statement specifying a format for data transfer. The format specified can be explicit (specified in a format specification) or implicit (specified using list-directed or namelist formatting). Contrast with unformatted I/O statement. See also list-directed I/O statement and namelist I/O statement.
frame window	The outermost parent window in QuickWin.

function	A series of statements that perform some operation and return a single value (through the function or result name) to the calling program unit. A function is invoked by a function reference in a main program unit or a subprogram unit. In Fortran 95/90, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol. The function is invoked by the appearance of the new or extended operator in the expression (along with the appropriate operands). For example, the symbol * can be defined for logical operands, extending its intrinsic definition for numeric operands. See also function subprogram, statement function, and subroutine.
function reference	Used in an expression to invoke a function, it consists of the function name and its actual arguments. A function reference returns a value (through the function or result name) that is used to evaluate the calling expression.
function result	The result value associated with a particular execution or call to a function. This result can be of any data type (including derived type) and can be array-valued. In a FUNCTION statement, the RESULT option can be used to give the result a name different from the function name. This option is required for a recursive function that directly calls itself.
function subprogram	A sequence of statements beginning with a FUNCTION (or optional OPTIONS) statement that is not in an interface block and ending with the corresponding END statement. See also function.

Glossary G

generic identifier	A generic name, operator, or assignment specified in an INTERFACE statement that is associated with all of the procedures within the interface block. Also called a generic specification.
global entity	An entity (a program unit, common block, or external procedure) that can be used with the same meaning throughout the executable program. A global entity has global scope; it is accessible throughout an executable program. See also local entity.
global section	A data structure (for example, global COMMON) or shareable image section potentially available to all processes in the system.

Glossary H

handle	A value (often, but not always, a 32-bit integer) that identifies some operating system resource, for example, a window or a process. The handle value is returned from an operating system call when the

	resource is created; your program then passes that value as an argument to subsequent operating system routines to identify which resource is being accessed. Your program should consider the handle value a "private" type and not try to interpret it as having any specific meaning (for example, an address).
hexadecimal constant	A constant that is a string of hexadecimal (base 16) digits (range 0 to 9, or an uppercase or lowercase letter in the range A to F) enclosed by apostrophes or quotation marks and preceded by the letter Z.
Hollerith constant	A constant that is a string of printable ASCII characters preceded by *n*H, where *n* is the number of characters in the string (including blanks and tabs).
host	Either the main program or subprogram that contains an internal procedure, or the module that contains a module procedure. The data environment of the host is available to the (internal or module) procedure.
host association	The process by which a module procedure, internal procedure, or derived-type definition accesses the entities of its host.

Glossary I

implicit interface	A procedure interface whose properties (the collection of names, attributes, and arguments of the procedure) are not known within the scope of the calling program, and have to be assumed. The information is assumed by the calling program from the properties of the procedure name and actual arguments in the procedure call.
implicit typing	The mechanism by which the data type for a variable is determined by the beginning letter of the variable name.
import library	A .LIB file that contains information about one or more dynamic-link libraries (DLLs), but does not contain the DLL's executable code. To provide the information needed to resolve the external references to DLL functions, the linker uses an import library when building an executable module of a process.
index	Can be either of the following: • The variable used as a loop counter in a DO statement. • An intrinsic function specifying the starting position of a substring inside a string.
initialize	The assignment of an initial value to a variable.

initialization expression	A form of constant expression that is used to specify an initial value for an entity.
inlining	An optimization that replaces a subprogram reference (CALL statement or function invocation) with the replicated code of the subprogram.
input/output (I/O)	The data that a program reads or writes. Also, devices to read and write data.
inquiry function	An intrinsic function whose result depends on properties of the principal argument, not the value of the argument.
integer constant	A constant that is a whole number with no decimal point. It can have a leading sign and is interpreted as a decimal number.
intent	An attribute of a dummy argument that is not a procedure or a pointer. It indicates whether the argument is used to transfer data into the procedure, out of the procedure, or both.
interactive process	A process that must periodically get user input to do its work. Contrast with background process.
interface	See procedure interface.
interface block	The sequence of statements starting with an INTERFACE statement and ending with the corresponding END INTERFACE statement.
interface body	The sequence of statements in an interface block starting with a FUNCTION or SUBROUTINE statement and ending with the corresponding END statement. Also called a procedure interface body.
internal file	The designated internal storage space (or variable buffer) that is manipulated during input and output. An internal file can be a character variable, character array, character array element, or character substring. In general, an internal file contains one record. However, an internal file that is a character array has one record for each array element.
internal procedure	A procedure (other than a statement function) that is contained within an internal subprogram. The program unit containing an internal procedure is called the host of the internal procedure. The internal procedure (which appears between a CONTAINS and END statement) is local to its host and inherits the host's environment through host association.
internal subprogram	A subprogram contained in a main program or another subprogram.
intrinsic	Describes entities defined by the Fortran 95/90 language (such as data types and procedures). Intrinsic entities can be used freely in any scoping unit.
intrinsic procedure	A subprogram supplied as part of the Fortran 95/90 library that performs array, mathematical, numeric, character, bit manipulation, and other miscellaneous functions. Intrinsic procedures are automatically available

	to any Fortran 95/90 program unit (unless specifically overridden by an EXTERNAL statement or a procedure interface block). Also called a built-in or library procedure.
invoke	To call upon; used especially with reference to subprograms. For example, to invoke a function is to execute the function.
iteration count	The number of executions of the DO range, which is determined as follows: `[(terminal value - initial value + increment value) / increment value]`

Glossary K

keyword	See argument keyword and statement keyword.
kind type parameter	Indicates the range of an intrinsic data type; for example: INTEGER(KIND=2). For real and complex types, it also indicates precision. If a specific kind parameter is not specified, the kind is the default for that type (for example, default integer). See also default character, default complex, default integer, default logical, and default real.

Glossary L

label	An integer, from 1 to 5 digits long, that precedes a statement and identifies it. For example, labels can be used to refer to a FORMAT statement or branch target statement.
language extension	An Intel® Fortran language element or interpretation that is not part of the Fortran 95 standard.
lexical token	A sequence of one or more characters that have an indivisible interpretation. A lexical token is the smallest meaningful unit (a basic language element) of a Fortran 95/90 statement; for example, constants, and statement keywords.
library routines	Files that contain functions, subroutines, and data that can be used by Fortran programs. For example: one library contains routines that handle the various differences between Fortran and C in argument passing and data types; another contains run-time functions and subroutines for Windows* graphics and QuickWin* applications. Some library routines are intrinsic (automatically available) to Fortran; others may require a specific USE statement to access the module defining the routines. See also intrinsic procedure.

line	A source form record consisting of 0 or more characters. A standard Fortran 95/90 line is limited to a maximum of 132 characters.
linker	A system program that creates an executable program from one or more object files produced by a language compiler or assembler. The linker resolves external references, acquires referenced library routines, and performs other processing required to create Linux* and Windows* executable files.
list-directed I/O statement	An implicit, formatted I/O statement that uses an asterisk (*) specifier rather than an explicit format specification. See also formatted I/O statement and namelist I/O statement.
listing	A printed copy of a program.
literal constant	A constant without a name; its value is directly specified in a program. See also named constant.
little endian	A method of data storage in which the least significant bit of a numeric value spanning multiple bytes is in the lowest addressed byte. This is the method used on Intel® systems. Contrast with big endian.
local entity	An entity that can be used only within the context of a subprogram (its scoping unit); for example, a statement label. A local entity has local scope. See also global entity.
local optimization	A level of optimization enabling optimizations within the source program unit and recognition of common expressions. See also optimization.
local symbol	A name defined in a program unit that is not accessible outside of that program unit.
logical constant	A constant that specifies the value .TRUE. or .FALSE..
logical expression	An integer or logical constant, variable, function value, or another constant expression, joined by a relational or logical operator. The logical expression is evaluated to a value of either true or false. For example, .NOT. 6.5 + (B .GT. D).
logical operator	A symbol that represents an operation on logical expressions. The logical operators are .AND., .OR., .NEQV., .XOR., .EQV., and .NOT..
logical unit	A channel in memory through which data transfer occurs between the program and the device or file. See also unit identifier.
longword	Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered 0 to 31. The address of the longword is the address of the byte containing bit 0. When the longword is interpreted as a signed integer, bit 31 is the sign bit. The value of signed integers is in the range -2**31 to 2**31-1. The value of unsigned integers is in the range 0 to 2**32-1.
loop	A group of statements that are executed repeatedly until an ending condition is reached.

lower bounds	See bounds.

Glossary M

main program	The first program unit to receive control when a program is run; it exercises control over subprograms. The main program usually contains a PROGRAM statement (or does not contain a SUBROUTINE, FUNCTION, or BLOCK DATA statement). Contrast with subprogram.
makefile	On Linux* and Mac OS* X systems, an argument to the make command containing a sequence of entries that specify dependencies. On Windows* systems, a file passed to the NMAKE utility containing a sequence of entries that specify dependencies. The contents of a makefile override the system built-in rules for maintaining, updating, and regenerating groups of programs. For more information on makefiles on Linux and Mac OS X systems, see make(1). For more information on using makefiles on Windows systems, see Building Applications: Makefile Command-Line Syntax.
many-one array section	An array section with a vector subscript having two or more elements with the same value.
master thread	In an OpenMP* Fortran program, the thread that creates a team of threads when a parallel region (PARALLEL directive construct) is encountered. The statements in the parallel region are then executed in parallel by each thread in the team. At the end of the parallel region, the team threads synchronize and only the master thread continues execution. See also thread.
message file	A Linux* and Mac OS* X catalog that contains the diagnostic message text of errors that can occur during program execution (run time).
metacommand	See compiler directive.
misaligned data	Data not aligned on a natural boundary. See also natural boundary.
module	A program unit that contains specifications and definitions that other program units can access (unless the module entities are declared PRIVATE). Modules are referenced in USE statements.
module procedure	A subroutine or function that is not an internal procedure and is contained in a module. The module procedure appears between a CONTAINS and END statement in its host module, and inherits the host module's environment through host association. A module procedure can be declared PRIVATE to the module; it is public by default.
multibyte character set	A character set in which each character is identified by using more than one byte. Although Unicode characters are 2 bytes wide, the Unicode character set is not referred to by this term.

multitasking	The ability of an operating system to execute several programs (tasks) at once.
multithreading	The ability of an operating system to execute different parts of a program, called threads, simultaneously. If the system supports parallel processing, multiple processors may be used to execute the threads.

Glossary N

name	Identifies an entity within a Fortran program unit (such as a variable, function result, common block, named constant, procedure, program unit, namelist group, or dummy argument). A name can contain letters, digits, underscores (_), and the dollar sign ($) special character. The first character must be a letter or a dollar sign. In FORTRAN 77, this term was called a symbolic name.
name association	Pertains to argument, host, or use association. See also argument association, host association, and use association.
named common block	A common block (one or more contiguous areas of storage) with a name. Common blocks are defined by a COMMON statement.
named constant	A constant that has a name. In FORTRAN 77, this term was called a symbolic constant.
namelist I/O statement	An implicit, formatted I/O statement that uses a namelist group specifier rather than an explicit format specifier. See also formatted I/O statement and list-directed I/O statement.
NaN	Not-a-Number. The condition that results from a floating-point operation that has no mathematical meaning; for example, zero divided by zero.
natural boundary	The virtual address of a data item that is the multiple of the size of its data type. For example, a REAL(8) (REAL*8) data item aligned on natural boundaries has an address that is a multiple of eight.
naturally aligned record	A record that is aligned on a hardware-specific natural boundary; each field is naturally aligned. (For more information, see *Optimizing Applications*: Setting Data Type and Alignment.) Contrast with packed record.
nesting	The placing of one entity (such as a construct, subprogram, format specification, or loop) inside another entity of the same kind. For example, nesting a loop within another loop (a nested loop), or nesting a subroutine within another subroutine (a nested subroutine).
nonexecutable statement	A Fortran 95/90 statement that describes program attributes, but does not cause any action to be taken when the program is executed.

nonsignaled	The state of an object used for synchronization in one of the wait functions is either signaled or nonsignaled. A nonsignaled state can prevent the wait function from returning. See also wait function.
numeric expression	A numeric constant, variable, or function value, or combination of these, joined by numeric operators and parentheses, so that the entire expression can be evaluated to produce a single numeric value. For example, -L or X+(Y-4.5*Z).
numeric operator	A symbol designating an arithmetic operation. In Fortran 95/90, the symbols +, -, *, /, and ** are used to designate addition, subtraction, multiplication, division, and exponentiation, respectively.
numeric storage unit	The unit of storage for holding a non-pointer scalar value of type default real, default integer, or default logical. One numeric storage unit corresponds to 4 bytes of memory.
numeric type	Integer, real, or complex type.

Glossary O

object	See data object.
object file	The binary output of a language processor (such as an assembler or compiler), which can either be executed or used as input to the linker.
obsolescent feature	A feature of FORTRAN 77 that is considered to be redundant in Fortran 95/90. These features are still in frequent use.
octal constant	A constant that is a string of octal (base 8) digits (range of 0 to 7) enclosed by apostrophes or quotation marks and preceded by the letter O.
operand	The passive element in an expression on which an operation is performed. Every expression must have at least one operand. For example, in I .NE. J, I and J are operands. Contrast with operator.
operation	A computation involving one or two operands.
operator	The active element in an expression that performs an operation. An expression can have zero or more operators. Intrinsic operators are arithmetic (+, -, *, /, and **) or logical (.AND., .NOT., and so on). For example, in I .NE. J, .NE. is the operator. Executable programs can define operators which are not intrinsic.
optimization	The process of producing efficient object or executing code that takes advantage of the hardware architecture to produce more efficient execution.

optional argument	A dummy argument that has the OPTIONAL attribute (or is included in an OPTIONAL statement in the procedure definition). This kind of argument does not have to be associated with an actual argument when its procedure is invoked.
order of subscript progression	A characteristic of a multidimensional array in which the leftmost subscripts vary most rapidly. Also called column-major order.
overflow	An error condition occurring when an arithmetic operation yields a result that is larger than the maximum value in the range of a data type.

Glossary P

packed record	A record that starts on an arbitrary byte boundary; each field starts in the next unused byte. Contrast with naturally aligned record.
pad	The filling of unused positions in a field or character string with dummy data (such as zeros or blanks).
parallel processing	The simultaneous use of more than one processor (CPU) to execute a program.
parameter	Can be either of the following: • In general, any quantity of interest in a given situation; often used in place of the term "argument". • A Fortran 95/90 named constant.
parent process	A process that initiates and controls another process (child). The parent process defines the environment for the child process. Also, the parent process can suspend or terminate without affecting the child process. See also child process.
parent window	A window that has one or more child windows. See also child window.
pathname	The path from the root directory to a subdirectory or file. See also root.
pipe	A connection that allows one program to get its input directly from the output of another program.
platform	A combination of operating system and hardware that provides a distinct environment in which to use a software product (for example, Windows* 2000 on processors using IA-32 architecture).
pointer	Is one of the following: • A Fortran 95/90 pointer

A data object that has the POINTER attribute. To be referenced or defined, it must be "pointer-associated" with a target (have storage space associated with it). If the pointer is an array, it must be pointer-associated to have a shape. See also pointer association.

- An integer pointer

  A data object that contains the address of its paired variable. This is also called a Cray* pointer.

pointer assignment — The association of a pointer with a target by the execution of a pointer assignment statement or the execution of an assignment statement for a data object of derived type having the pointer as a subobject.

pointer association — The association of storage space to a Fortran 95/90 pointer by means of a target. A pointer is associated with a target after pointer assignment or the valid execution of an ALLOCATE statement.

precision — The number of significant digits in a real number. See also double-precision constant, kind type parameter, and single-precision constant.

primary — The simplest form of an expression. A primary can be any of the following data objects:

- A constant
- A constant subobject (parent is a constant)
- A variable (scalar, structure, array, or pointer; an array cannot be assumed size)
- An array constructor
- A structure constructor
- A function reference
- An expression in parentheses

primary thread — The initial thread of a process. Also called the main thread or thread 1. See also thread.

procedure — A computation that can be invoked during program execution. It can be a subroutine or function, an internal, external, dummy or module procedure, or a statement function. A subprogram can define more than one procedure if it contains an ENTRY statement. See also subprogram.

procedure interface	The statements that specify the name and characteristics of a procedure, the name and characteristics of each dummy argument, and the generic identifier (if any) by which the procedure can be referenced. The characteristics of a procedure are fixed, but the remainder of the interface can change in different scoping units. If these properties are all known within the scope of the calling program, the procedure interface is explicit; otherwise it is implicit (deduced from its reference and declaration).
process object	A virtual address space, security profile, a set of threads that execute in the address space of the process, and a set of resources visible to all threads executing in the process. Several thread objects can be associated with a single process.
program	A set of instructions that can be compiled and executed by itself. Program blocks contain a declaration and an executable section.
program section	A particular common block or local data area for a particular routine containing equivalence groups.
program unit	The fundamental component of an executable program. A sequence of statements and comment lines. It can be a main program, a module, an external subprogram, or a block data program unit.

Glossary Q

quadword	Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered 0 to 63. (Bit 63 is used as the sign bit.) A quadword is identified by the address of the word containing the low-order bit (bit 0). The value of a signed quadword integer is in the range -2**63 to 2**63-1.

Glossary R

random access	See direct access.
rank	The number of dimensions of an array. A scalar has a rank of zero.
rank-one object	A data structure comprising scalar elements with the same data type and organized as a simple linear sequence. See also scalar.
real constant	A constant that is a number written with a decimal point, exponent, or both. It can have single precision (REAL(KIND=4)), double precision (REAL(KIND=8)), or quad precision (REAL(KIND=16)).
record	Can be either of the following:

	• A set of logically related data items (in a file) that is treated as a unit; such a record contains one or more fields. This definition applies to I/O records and items that are declared in a record structure. • One or more data items that are grouped in a structure declaration and specified in a RECORD statement.
record access	The method used to store and retrieve records in a file.
record structure declaration	A block of statements that define the fields in a record. The block begins with a STRUCTURE statement and ends with END STRUCTURE. The name of the structure must be specified in a RECORD statement.
record type	The property that determines whether records in a file are all the same length, of varying length, or use other conventions to define where one record ends and another begins.
recursion	Pertains to a subroutine or function that directly or indirectly references itself.
reference	Can be any of the following: • For a data object, the appearance of its name, designator, or associated pointer where the value of the object is required. When an object is referenced, it must be defined. • For a procedure, the appearance of its name, operator symbol, or assignment symbol that causes the procedure to be executed. Procedure reference is also called "calling" or "invoking" a procedure. • For a module, the appearance of its name in a USE statement.
relational expression	An expression containing one relational operator and two operands of numeric or character type. The result is a value that is true or false. For example, A-C .GE. B+2 or DAY .EQ. 'MONDAY'.
relational operator	The symbols used to express a relational condition or expression. The relational operators are (.EQ., .NE., .LT., .LE., .GT., and .GE.).
relative file organization	A file organization that consists of a series of component positions, called cells, numbered consecutively from 1 to n. Intel Fortran uses these numbered, fixed-length cells to calculate the component's physical position in the file.
relative pathname	A directory path expressed in relation to any directory other than the root directory. Contrast with absolute pathname.
root	On Windows* systems, the top-level directory on a disk drive; it is represented by a backslash (\). For example, C:\ is the root directory for drive C. On Linux* systems, the top-level directory in the file system; it is represented by a slash (/).

routine	A subprogram; a function or procedure. See also function, subroutine, and procedure.
run time	The time during which a computer executes the statements of a program.

Glossary S

saved object	A variable that retains its association status, allocation status, definition status, and value after execution of a RETURN or END statement in the scoping unit containing the declaration.
scalar	Pertaining to data items with a rank of zero. A single data object of any intrinsic or derived data type. Contrast with array. See also rank-one object.
scalar memory reference	A reference to a scalar variable, scalar record field, or array element that resolves into a single data item (having a data type) and can be assigned a value with an assignment statement. It is similar to a scalar reference, but it excludes constants, character substrings, and expressions.
scalar reference	A reference to a scalar variable, scalar record field, derived-type component, array element, constant, character substring, or expression that resolves into a single data item having a data type.
scalar variable	A variable name specifying one storage location.
scale factor	A number indicating the location of the decimal point in a real number and, if there is no exponent, the size of the number on input.
scope	The portion of a program in which a declaration or a particular name has meaning. Scope can be global (throughout an executable program), scoping unit (local to the scoping unit), or statement (within a statement, or part of a statement).
scoping unit	The part of the program in which a name has meaning. It is one of the following: • A program unit or subprogram • A derived-type definition • A procedure interface body Scoping units cannot overlap, though one scoping unit can contain another scoping unit. The outer scoping unit is called the host scoping unit.
screen coordinates	Coordinates relative to the upper left corner of the screen.

section subscript	A subscript list (enclosed in parentheses and appended to the array name) indicating a portion (section) of an array. At least one of the subscripts in the list must be a subscript triplet or vector subscript. The number of section subscripts is the rank of the array. See also array section, subscript, subscript triplet, and vector subscript.
seed	A value (which can be assigned to a variable) that is required in order to properly determine the result of a calculation; for example, the argument i in the random number generator (RAN) function syntax: `y = RAN (i)`
selector	A mechanism for designating the following: • Part of a data object (an array element or section, a substring, a derived type, or a structure component) • The set of values for which a CASE block is executed
sequence	A set ordered by a one-to-one correspondence with the numbers 1 through n, where *n* is the total number of elements in the sequence. A sequence can be empty (contain no elements).
sequential access	A method for retrieving or storing data in which the data (record) is read from, written to, or removed from a file based on the logical order (sequence) of the record in the file. (The record cannot be accessed directly.) Contrast with direct access.
sequential file organization	A file organization in which records are stored one after the other, in the order in which they were written to the file.
shape	The rank and extents of an array. Shape can be represented by a rank-one array (vector) whose elements are the extents in each dimension.
shape conformance	Pertains to the rule concerning operands of binary intrinsic operations in expressions: to be in shape conformance, the two operands must both be arrays of the same shape, or one or both of the operands must be scalars.
short field termination	The use of a comma (,) to terminate the field of a numeric data edit descriptor. This technique overrides the field width (w) specification in the data edit descriptor and therefore avoids padding of the input field. The comma can only terminate fields less than w characters long. See also data edit descriptor.
signal	The software mechanism used to indicate that an exception condition (abnormal event) has been detected. For example, a signal can be generated by a program or hardware error, or by request of another program.

single-precision constant	A processor approximation of the value of a real number that occupies 4 bytes of memory and can assume a positive, negative, or zero value. The precision is less than a constant of double-precision type. For the precise ranges of the single-precision constants, see Building Applications: Data Representation Overview. See also denormalized number.
size	The total number of elements in an array (the product of the extents).
source file	A program or portion of a program library, such as an object file, or image file.
specification expression	A restricted expression that is of type integer and has a scalar value. This type of expression appears only in the declaration of array bounds and character lengths.
specification statement	A nonexecutable statement that provides information about the data used in the source program. Such a statement can be used to allocate and initialize variables, arrays, records, and structures, and define other characteristics of names used in a program.
statement	An instruction in a programming language that represents a step in a sequence of actions or a set of declarations. In Fortran 95/90, an ampersand can be used to continue a statement from one line to another, and a semicolon can be used to separate several statements on one line. There are two main classes of statements: executable and nonexecutable.
statement function	A computing procedure defined by a single statement in the same program unit in which the procedure is referenced.
statement function definition	A statement that defines a statement function. Its form is the statement function name (followed by its optional dummy arguments in parentheses), followed by an equal sign (=), followed by a numeric, logical, or character expression. A statement function definition must precede all executable statements and follow all specification statements.
statement keyword	A word that begins the syntax of a statement. All program statements (except assignment statements and statement function definitions) begin with a statement keyword. Examples are INTEGER, DO, IF, and WRITE.
statement label	See label.
static variable	A variable whose storage is allocated for the entire execution of a program.

storage association	The relationship between two storage sequences when the storage unit of one is the same as the storage unit of the other. Storage association is provided by the COMMON and EQUIVALENCE statements. For modules, pointers, allocatable arrays, and automatic data objects, the SEQUENCE statement defines a storage order for structures.
storage location	An addressable unit of main memory.
storage sequence	A sequence of any number of consecutive storage units. The size of a storage sequence is the number of storage units in the storage sequence. A sequence of storage sequences forms a composite storage sequence. See also storage association and storage unit.
storage unit	In a storage sequence, the number of storage units needed to represent one real, integer, logical, or character value. See also character storage unit, numeric storage unit, and storage sequence.
stride	The increment between subscript values that can optionally be specified in a subscript triplet. If it is omitted, it is assumed to be one.
string edit descriptor	A format descriptor that transfers characters to an output record.
structure	Can be either of the following: • A scalar data object of derived (user-defined) type. • An aggregate entity containing one or more fields or components.
structure component	Can be either of the following: • One of the components of a structure. • An array whose elements are components of the elements of an array of derived type.
structure constructor	A mechanism that is used to specify a scalar value of a derived type. A structure constructor is the name of the type followed by a parenthesized list of values for the components of the type.
subobject	Part of a data object (parent object) that can be referenced and defined separately from other parts of the data object. A subobject can be an array element, an array section, a substring, a derived type, or a structure component. Subobjects are referenced by designators and can be considered to be data objects themselves. See also designator.
subobject designator	See designator.
subprogram	A function or subroutine subprogram that can be invoked from another program unit to perform a specific task. A subprogram can define more than one procedure if it contains an ENTRY statement. Contrast with main program. See also procedure.

subroutine	A procedure that can return many values, a single value, or no value to the calling program unit (through arguments). A subroutine is invoked by a CALL statement in another program unit. In Fortran 95/90, a subroutine can also be used to define a new form of assignment (defined assignment), which is different from those intrinsic to Fortran 90. Such assignments are invoked with an ASSIGNMENT(=) interface block rather than the CALL statement. See also function, statement function, and subroutine subprogram.
subroutine subprogram	A sequence of statements starting with a SUBROUTINE (or optional OPTIONS) statement and ending with the corresponding END statement. See also subroutine.
subscript	A scalar integer expression (enclosed in parentheses and appended to the array name) indicating the position of an array element. The number of subscripts is the rank of the array. See also array element.
subscript triplet	An item in a section subscript list specifying a range of values for the array section. A subscript triplet contains at least one colon and has three optional parts: a lower bound, an upper bound, and a stride. Contrast with vector subscript. See also array section and section subscript.
substring	A contiguous portion of a scalar character string. Do not confuse this with the substring selector in an array section, where the result is another array section, not a substring.
symbolic name	See name.
syntax	The formal structure of a statement or command string.

Glossary T

target	The named data object associated with a pointer (in the form pointer-object => target). A target is declared in a type declaration statement that contains the TARGET attribute. See also pointer and pointer association.
thread	Part of a program that can run at the same time as other parts, usually with some form of communication and/or synchronization among the threads. See also multithreading.
transformational function	An intrinsic function that is not an elemental or inquiry function. A transformational function usually changes an array actual argument into a scalar result or another array, rather than applying the argument element by element.
truncation	Can be either of the following:

- A technique that approximates a numeric value by dropping its fractional value and using only the integer portion.
- The process of removing one or more characters from the left or right of a number or string.

type declaration statement	A nonexecutable statement specifying the data type of one or more variables: an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, CHARACTER, LOGICAL, or TYPE statement. In Fortran 95/90, a type declaration statement may also specify attributes for the variables. Also called a type declaration or type specification.
type parameter	Defines an intrinsic data type. The type parameters are kind and length. The kind type parameter (KIND=) specifies the range for the integer data type, the precision and range for real and complex data types, and the machine representation method for the character and logical data types. The length type parameter (LEN=) specifies the length of a character string. See also kind type parameter.

Glossary U

ultimate component	For a derived type or a structure, a component that is of intrinsic type or has the POINTER attribute, or an ultimate component of a component that is a derived type and does not have the POINTER attribute.
unary operator	An operator that operates on one operand. For example, the minus sign in -A and the .NOT. operator in .NOT. (J .GT. K).
undefined	For a data object, the property of not having a determinate value.
underflow	An error condition occurring when the result of an arithmetic operation yields a result that is smaller than the minimum value in the range of a data type. For example, in unsigned arithmetic, underflow occurs when a result is negative. See also denormalized number.
unformatted data	Data written to a file by using unformatted I/O statements; for example, binary numbers.
unformatted I/O statement	An I/O statement that does not contain format specifiers and therefore does not translate the data being transferred. Contrast with formatted I/O statement.
unformatted record	A record that is transmitted in internal format between internal and external storage.
unit identifier	The identifier that specifies an external unit or internal file. The identifier can be any one of the following:

- An integer expression whose value must be zero or positive

	• An asterisk (*) that corresponds to the default (or implicit) I/O unit • The name of a character scalar memory reference or character array name reference for an internal file Also called a device code, or logical unit number.
unspecified storage unit	A unit of storage for holding a pointer or a scalar that is not a pointer and is of type other than default integer, default character, or default real.
upper bounds	See bounds.
use association	The process by which the entities in a module are made accessible to other scoping units (through a USE statement in the scoping unit).
user-defined assignment	See defined assignment.
user-defined operator	See defined operation.
user-defined type	See derived type.

Glossary V

variable	A data object (stored in a memory location) whose value can change during program execution. A variable can be a named data object, an array element, an array section, a structure component, or a substring. In FORTRAN 77, a variable was always scalar and named. Contrast with constant.
variable format expression	A numeric expression enclosed in angle brackets (<>) that can be used in a FORMAT statement. If necessary, it is converted to integer type before use.
variable-length record type	A file format in which records may be of different lengths.
vector subscript	A rank-one array of integer values used as a section subscript to select elements from a parent array. Unlike a subscript triplet, a vector subscript specifies values (within the declared bounds for the dimension) in an arbitrary order. Contrast with subscript triplet. See also array section and section subscript.

Glossary W

wait function	A function that blocks the execution of a calling thread until a specified set of conditions has been satisfied.

whole array	An array reference (for example, in a type declaration statement) that consists of the array name alone, without subscript notation. Whole array operations affect every element in the array. See also array.

Glossary Z

zero-sized array	An array with at least one dimension that has at least one extent of zero. A zero-sized array has a size of zero and contains no elements. See also array.

Index

_
    in names 1748
__INTEL_COMPILER_BUILD_DATE symbol 153
__INTEL_COMPILER symbol 153
_DLL symbol 153
_FTN _ALLOC 116
_M_AMD64 symbol 153
_M_IA64 symbol 153
_M_IX86 symbol 153
_M_X64 symbol 153
_MT symbol 153
_OPENMP symbol 153
_PGO_INSTRUMENT symbol 153
_VF_VER_ symbol 153
_WIN32 symbol 153
_WIN64 symbol 153
,
    as external field separator 2031
    using to separate input data 2066
;
    as source statement separator 1752
:
    in array specifications 1809, 1853, 1857, 1859, 1860
!
    as comment indicator 1757
!DEC$ 2159
/
    in slash editing 2077
/? compiler option 663
// 1823, 2473
/= 1823
/1 compiler option 764, 935
/4I2 compiler option 691
/4I4 compiler option 691
/4I8 compiler option 691
/4L132 compiler option 565
/4L72 compiler option 565
/4L80 compiler option 565
/4Na compiler option 498
/4Naltparam compiler option 478
/4Nb compiler option 511
/4Nd compiler option 1099
/4Nf compiler option 588
/4Nportlib compiler option 469, 471
/4Ns compiler option 1063
/4R16 compiler option 1046
/4R8 compiler option 1046
/4Ya compiler option 498
/4Yaltparam compiler option 478
/4Yb compiler option 511
/4Yd compiler option 1099
/4Yf compiler option 638
/4Yportlib compiler option 469, 471
/4Ys compiler option 1063
/align compiler option 472
/allow
    fpp_comments compiler option 476
/altparam compiler option 478
/arch compiler option 480
/architecture compiler option 480
/asmattr
    all compiler option 483
    machine compiler option 483
    none compiler option 483
    source compiler option 483
/asmfile compiler option 485
/assume
    bscc compiler option 486

/assume *(continued)*
buffered_io compiler option 486
byterecl compiler option 486
cc_omp compiler option 486
dummy_aliases compiler option 486
ieee_fpe_flags compiler option 486
minus0 compiler option 486
none compiler option 486
old_boz compiler option 486
old_logical_ldio compiler option 486
old_maxminloc compiler option 486
old_unit_star compiler option 486
old_xor compiler option 486
protect_constants compiler option 486
protect_parens compiler option 486
realloc_lhs compiler option 486
source_include compiler option 486
std_mod_proc_name compiler option 486
underscore compiler option 486
/auto compiler option 498
/automatic compiler option 498
/bigobj compiler option 506
/bintext compiler option 507
/CB compiler option 511
/ccdefault
default compiler option 510
fortran compiler option 510
list compiler option 510
/c compiler option 509
/C compiler option 511
/check
all compiler option 511
arg_temp_created compiler option 511
bounds compiler option 511
none compiler option 511
output_conversion compiler option 511
uninit compiler option 511
/check compiler option 511
/compile-only compiler option 509
/convert
big_endian compiler option 517
cray compiler option 517
fdx compiler option 517
fgx compiler option 517
ibm compiler option 517
little_endian compiler option 517
native compiler option 517
vaxd compiler option 517
vaxg compiler option 517
/CU compiler option 511
/dbglibs compiler option 524
/D compiler option 522
/debug compiler option 529
/debug-parameters
all compiler option 532
none compiler option 532
used compiler option 532
/define compiler option 522
/d-lines compiler option 523, 856
/dll compiler option 553
/double-size compiler option 554
/E compiler option 561
/EP compiler option 562
/error-limit compiler option 547, 871
/exe compiler option 563
/extend-source compiler option 565
/extfor compiler option 566
/extfpp compiler option 567
/extlnk compiler option 567
/f66 compiler option 570
/f77rtl compiler option 572
/Fa compiler option 573
/FA compiler option 573
/fast compiler option 577
/F compiler option 569
/Fe compiler option 563, 581
/FI compiler option 588
/fixed compiler option 588
/floating divide-by-zero 1714
/floating invalid 1714
/floating overflow 1714
/floating underflow 1714
/fltconsistency compiler option 590
/Fm compiler option 593
/Fo compiler option 600
/fp compiler option 601, 606, 1677, 1684
how to use 1684
/fpconstant compiler option 616
/fpe-all compiler option 620
/fpe compiler option 617, 1714
/fpp compiler option 625, 889, 1232
fpp options you can specify by using 1232
/fpscomp
all compiler option 627
filesfromcmd compiler option 627
general compiler option 627
ioformat compiler option 627
ldio_spacing compiler option 627

/fpscomp *(continued)*
libs compiler option 627
logicals compiler option 627
none compiler option 627
/fpscomp compiler option 627
/FR compiler option 638
/free compiler option 638
/G2 compiler option 651
/G2-p9000 compiler option 651
/G5 compiler option 653
/G6 compiler option 653
/G7 compiler option 653
/GB compiler option 653
/Ge compiler option 656
/gen-interfaces compiler option 657
/Gm compiler option 660, 670
/Gs compiler option 660
/GS compiler option 640, 641, 661
/Gz compiler option 662, 670
/heap-arrays compiler option 662
/help compiler option 663
/homeparams compiler option 665
/hotpatch compiler option 666
/I compiler option 667
/iface
cref compiler option 670
cvf compiler option 670
default compiler option 670
mixed_str_len_arg compiler option 670
stdcall compiler option 670
stdref compiler option 670
/iface compiler option 670
/include compiler option 667
/inline
all compiler option 674
manual compiler option 674
none compiler option 674
size compiler option 674
speed compiler option 674
/intconstant compiler option 690
/integer-size compiler option 691
/LD compiler option 553, 710
/libdir
all compiler option 710
automatic compiler option 710
none compiler option 710
user compiler option 710
/libdir compiler option 710
/libs
dll compiler option 712
qwin compiler option 712
qwins compiler option 712
static compiler option 712
/link compiler option 715
/logo compiler option 716
/map compiler option 720
/MD compiler option 726
/MDd compiler option 726
/MDs compiler option 712, 728
/MDsd compiler option 712, 728
/MG compiler option 1108
/ML compiler option 712, 733
/MLd compiler option 712, 733
/module compiler option 734
/MP compiler option 735, 743
/MT compiler option 739
/MTd compiler option 739
/MW compiler option 712
/MWs compiler option 712
/names
as_is compiler option 744
lowercase compiler option 744
uppercase compiler option 744
/nbs compiler option 486
/nodefine compiler option 522
/noinclude compiler option 1116
/Oa compiler option 573
/Ob compiler option 680, 758
/object compiler option 760
/O compiler option 753
/Od compiler option 761
/Og compiler option 763
/Op compiler option 590
/optimize compiler option 753
/Os compiler option 800
/Ot compiler option 802
/Ow compiler option 582
/Ox compiler option 753
/Oy compiler option 598, 600, 803
/P compiler option 827
/pdbfile compiler option 822
/preprocess-only compiler option 827
/Qansi-alias compiler option 479, 847
/Qauto_scalar compiler option 496, 848
/Qauto compiler option 498
/Qautodouble compiler option 1046
/Qax compiler option 500, 850, 1310

/Qchkstk compiler option 853
/Qcommon-args compiler option 486
/Qcomplex-limited-range compiler option 516, 855
/Qcpp compiler option 625, 889
/Qdiag compiler option 533, 539, 857, 863
/Qdiag-disable compiler option 533, 539, 857, 863
/Qdiag-dump compiler option 538, 862
/Qdiag-enable
    sc compiler option 533, 539, 857, 863
    sc-include compiler option 544, 867
    sc-parallel compiler option 545, 869
    sv-include compiler option 544, 867
/Qdiag-enable compiler option 533, 539, 857, 863
/Qdiag-error compiler option 533, 539, 857, 863
/Qdiag-error-limit compiler option 547, 871
/Qdiag-file-append compiler option 550, 873
/Qdiag-file compiler option 548, 872
/Qdiag-id-numbers compiler option 551, 875
/Qdiag-once compiler option 552, 876
/Qdiag-remark compiler option 533, 539, 857, 863
/Qdiag-warning compiler option 533, 539, 857, 863
/Qd-lines compiler option 523, 856
/Qdps compiler option 478
/Qdyncom compiler option 560, 877
/Qextend-source compiler option 565
/Qfast-transcendentals compiler option 578, 879
/Qfma compiler option 593, 880
/Qfnalign compiler option 574, 882
/Qfnsplit compiler option 597, 883
/Qfpp compiler option 625, 889
/Qfp-port compiler option 611, 884
/Qfp-relaxed compiler option 612, 885
/Qfp-speculation compiler option 613, 886
/Qfp-stack-check compiler option 615, 888
/Qftz compiler option 643, 891, 1689, 1714
/Qglobal-hoist compiler option 658, 893
/QIA64-fr32 compiler option 894
/QIfist compiler option 1008, 1044
/Qimsl compiler option 895
/Qinline-debug-info compiler option 676, 896
/Qinline-dllimport compiler option 897
/Qinline-factor compiler option 677, 898
/Qinline-forceinline compiler option 679, 900
/Qinline-max-per-compile compiler option 682, 901
/Qinline-max-per-routine compiler option 683, 903
/Qinline-max-size compiler option 685, 905
/Qinline-max-total-size compiler option 687, 906
/Qinline-min-size compiler option 688, 908
/Qinstruction compiler option 730, 911
/Qinstrument-functions compiler option 586, 912
/Qip compiler option 693, 914
/QIPF-fltacc compiler option 698, 919
/QIPF-flt-eval-method0 compiler option 696, 917
/QIPF-fma compiler option 593, 880
/QIPF-fp-relaxed compiler option 612, 885
/Qip-no-inlining compiler option 694, 915
/Qip-no-pinlining compiler option 695, 916
/Qipo-c compiler option 701, 922
/Qipo compiler option 699, 920, 1501
/Qipo-jobs compiler option 702, 923
/Qipo-S compiler option 704, 925
/Qipo-separate compiler option 705, 926
/Qivdep-parallel compiler option 707, 927
/Qkeep-static-consts compiler option 589, 928
/Qlocation compiler option 929
/Qlowercase compiler option 744
/Qmap-opts compiler option 721, 931
/Qmkl compiler option 732, 933
/Qnobss-init compiler option 746, 934
/Qonetrip compiler option 764, 935
/Qopenmp compiler option 765, 936
/Qopenmp-lib compiler option 766, 937, 1321
/Qopenmp-link compiler option 768, 939
/Qopenmp-profile compiler option 770, 940
/Qopenmp-report compiler option 771, 942
/Qopenmp-stubs compiler option 772, 943
/Qopenmp-threadprivate compiler option 774, 944
/Qopt-block-factor compiler option 775, 946
/Qoption compiler option 969
/Qopt-jump-tables compiler option 776, 947
/Qopt-loadpair compiler option 778, 948
/Qopt-mem-bandwidth compiler option 780, 949
/Qopt-mod-versioning compiler option 782, 951
/Qopt-multi-version-aggressive compiler option 783, 952
/Qopt-prefetch compiler option 784, 953
/Qopt-prefetch-initial-values compiler option 786, 955
/Qopt-prefetch-issue-excl-hint compiler option 787, 956
/Qopt-prefetch-next-iteration compiler option 788, 957
/Qopt-ra-region-strategy compiler option 790, 959, 1606
    example 1606
/Qopt-report compiler option 791, 960, 1260
/Qopt-report-file compiler option 793, 962
/Qopt-report-help compiler option 794, 963
/Qopt-report-phase compiler option 795, 964
/Qopt-report-routine compiler option 796, 965
/Qopt-streaming-stores compiler option 797, 966

/Qopt-subscript-in-range compiler option 799, 968
/Qpad compiler option 806, 971
/Qpad-source compiler option 807, 972
/Qpar-adjust-stack compiler option 974
/Qpar-affinity compiler option 808, 975
/Qparallel compiler option 819, 986
/Qpar-num-threads compiler option 810, 977
/Qpar-report compiler option 811, 978
/Qpar-schedule compiler option 814, 980
/Qpar-threshold compiler option 818, 984
/Qpc compiler option 821, 987
/Qprec compiler option 737, 989
/Qprec-div compiler option 825, 990
/Qprec-sqrt compiler option 826, 991
/Qprof-data-order compiler option 829, 992
/Qprof-dir compiler option 830, 994
/Qprof-file compiler option 832, 995
/Qprof-func-order compiler option 834, 996
/Qprof-gen
    srcpos compiler option 1530, 1532, 1552
/Qprof-gen:srcpos compiler option
    code coverage tool 1532
    test priorization tool 1552
/Qprof-gen compiler option 836, 998, 1530
/Qprof-genx compiler option 836, 998
/Qprof-hotness-threshold compiler option 838, 1000
/Qprof-src-dir compiler option 840, 1001
/Qprof-src-root compiler option 841, 1003
/Qprof-src-root-cwd compiler option 843, 1005
/Qprof-use compiler option 845, 1006, 1532, 1561
    code coverage tool 1532
    profmerge utility 1561
/Qrcd compiler option 1008, 1044
/Qrct compiler option 1009, 1045
/Qsafe-cray-ptr compiler option 1010, 1051
/Qsave compiler option 1012, 1053
/Qsave-temps compiler option 1013, 1054
/Qscalar-rep compiler option 1015, 1056
/Qsfalign compiler option 1016
/Qsox compiler option 1017, 1062
/Qtcheck compiler option 1019, 1072
/Qtcollect compiler option 1020, 1073
/Qtcollect-filter compiler option 1021, 1075
/Qtprofile compiler option 1023, 1078
/Qtrapuv compiler option 642, 1024
/Qunroll-aggressive compiler option 1027, 1086
/Qunroll compiler option 1026, 1085
/Quppercase compiler option 744
/Quse-asm compiler option 1028, 1088
/Quse-msasm-symbols compiler option 1029
/Quse-vcdebug compiler option 1030
/Qvc compiler option 1031
/Qvec compiler option 1032, 1090
/Qvec-guard-write compiler option 1033, 1091
/Qvec-report compiler option 1034, 1092
/Qvec-threshold compiler option 1036, 1094
/Qvms compiler option 1095
/Qx compiler option 1038, 1112, 1306
/Qzero compiler option 1042, 1121
/real-size compiler option 1046
/recursive compiler option 1047
/reentrancy
    async compiler option 1049
    none compiler option 1049
    threaded compiler option 1049
/RTCu compiler option 511
/S compiler option 1050
/source compiler option 1061
/stand
    f03 compiler option 1063
    f90 compiler option 1063
    f95 compiler option 1063
    none compiler option 1063
/stand compiler option 1063
/static compiler option 1065
/syntax-only compiler option 1070
/Tf compiler option 1061
/threads compiler option 1077
/traceback compiler option 1080
/u compiler option 1083
/U compiler option 1084
/undefine compiler option 1084
/us compiler option 486
/V compiler option 1090
/vms compiler option 1095
/W0 compiler option 1099
/W1 compiler option 1099
/warn
    alignments compiler option 1099
    all compiler option 1099
    declarations compiler option 1099
    errors compiler option 1099
    general compiler option 1099
    ignore_loc compiler option 1099
    interfaces compiler option 1099
    none compiler option 1099
    stderrors compiler option 1099
    truncated_source compiler option 1099

/warn *(continued)*
uncalled compiler option 1099
unused compiler option 1099
usage compiler option 1099
/warn compiler option 1099
/watch
all compiler option 1105
cmd compiler option 1105
none compiler option 1105
source compiler option 1105
/watch compiler option 1105
/WB compiler option 1106
/w compiler option 1098, 1099
/what compiler option 1107
/winapp compiler option 1108
/X compiler option 1116
/Z7 compiler option 529, 650, 1119, 1122
/Zd compiler option 529, 1121
/Zi compiler option 529, 650, 1119, 1122
/Zl compiler option 710
/Zp compiler option 472, 1124
/Zs compiler option 1070, 1124
/Zx compiler option 1124
.a files 261
.AND. 1825
.asm files 121
.def files 121
.DLL files 121, 261
.dpi file 1532, 1552, 1561
.dylib files 261
.dyn file 1532, 1552, 1561
.dyn files 1520, 1530
.EQ. 1823
.EQV. 1825
.EXE files 121, 257
creating 257
.f90 files 121
.f files 121
.for files 121, 257
.fpp files 121
.GE. 1823
.GT. 1823
.i90 files 121
.i files 121
.LE. 1823
.lib files 261
.LT. 1823
.MAP files 103
.MOD files 258
.NE. 1823
.NEQV. 1825
.NOT. 1825
.obj files 121, 257
.o files 121
.OR. 1825
.rbj files 121
.RES files 121
.so files 261
.spi file 1532, 1552
.XOR. 1825
(/.../) 1812
[...] 1812
$
implicit type in names 1800
in names 1748
*
as comment indicator 1757
in CHARACTER statements 1849
in format specifier 1984
in unit specifier 197, 1983
\ editing 2079
%
in non-Fortran procedures 1935
%LOC 2978, 3166
using with integer pointers 3166
%REF 3381
%VAL 3651
1823
1823
== 1823
=> 1840
> 1823
>= 1823
-132 compiler option 565
-1 compiler option 764, 935
-66 compiler option 570
-72 compiler option 565
-80 compiler option 565
-align compiler option 472
-allow fpp_comments compiler option 476
-altparam compiler option 478
-ansi-alias compiler option 479, 847
-arch compiler option 480
-assume 2underscores compiler option 486
-assume bscc compiler option 486
-assume buffered_io compiler option 486
-assume byterecl compiler option 486
-assume cc_omp compiler option 486

-assume dummy_aliases compiler option 486
-assume ieee_fpe_flags compiler option 486
-assume minus0 compiler option 486
-assume none compiler option 486
-assume old_boz compiler option 486
-assume old_logical_ldio compiler option 486
-assume old_maxminloc compiler option 486
-assume old_unit_star compiler option 486
-assume old_xor compiler option 486
-assume protect_constants compiler option 486
-assume protect_parens compiler option 486
-assume realloc_lhs compiler option 486
-assume source_include compiler option 486
-assume std_mod_proc_name compiler option 486
-assume underscore compiler option 486
-auto compiler option 498
-autodouble compiler option 1046
-automatic compiler option 498
-auto-scalar compiler option 496, 848
-ax compiler option 500, 850, 1310
-B compiler option 503
-Bdynamic compiler option (Linux* only) 504
-Bstatic compiler option (Linux* only) 508
-CB compiler option 511
-ccdefault default compiler option 510
-ccdefault fortran compiler option 510
-ccdefault list compiler option 510
-c compiler option 509
-C compiler option 511
-check all compiler option 511
-check arg_temp_created compiler option 511
-check bounds compiler option 511
-check compiler option 511
-check none compiler option 511
-check output_conversion compiler option 511
-check pointers compiler option 511
-check uninit compiler option 511
-cm compiler option 1099
-common-args compiler option 486
-complex-limited-range compiler option 516, 855
-convert big_endian compiler option 517
-convert cray compiler option 517
-convert fdx compiler option 517
-convert fgx compiler option 517
-convert ibm compiler option 517
-convert little_endian compiler option 517
-convert native compiler option 517
-convert vaxd compiler option 517
-convert vaxg compiler option 517
-cpp compiler option 625, 889
-CU compiler option 511
-cxxlib compiler option 520
-cxxlib-gcc compiler option 520
-cxxlib-nostd compiler option 520
-D compiler option 522
-DD compiler option 523, 856
-debug compiler option 526
-debug-parameters all compiler option 532
-debug-parameters none compiler option 532
-debug-parameters used compiler option 532
-diag compiler option 533, 539, 857, 863
-diag-disable compiler option 533, 539, 857, 863
-diag-dump compiler option 538, 862
-diag-enable compiler option 533, 539, 857, 863
-diag-enable sc compiler option 533, 539, 857, 863
-diag-enable sc-include compiler option 544, 867
-diag-enable sc-parallel compiler option 545, 869
-diag-enable sv-include compiler option 544, 867
-diag-error compiler option 533, 539, 857, 863
-diag-error-limit compiler option 547, 871
-diag-file-append compiler option 550, 873
-diag-file compiler option 548, 872
-diag-id-numbers compiler option 551, 875
-diag-once compiler option 552, 876
-diag-remark compiler option 533, 539, 857, 863
-diag-warning compiler option 533, 539, 857, 863
-d-lines compiler option 523, 856
-double-size compiler option 554
-dps compiler option 478
-dryrun compiler option 556
-dynamiclib compiler option (Mac OS* X only) 559
-dynamic-linker compiler option (Linux* only) 558
-dyncom compiler option 560, 877
-e03 compiler option 1099
-e90 compiler option 1099
-e95 compiler option 1099
-E compiler option 561
-EP compiler option 562
-error-limit compiler option 547, 871
-extend-source compiler option 565
-f66 compiler option 570
-f77rtl compiler option 572
-falias compiler option 573
-falign-functions compiler option 574, 882
-falign-stack compiler option 575
-fast compiler option 577
-fast-transcendentals compiler option 578, 879
-fcode-asm compiler option 580

-fexceptions compiler option 581
-ffnalias compiler option 582
-FI compiler option 588
-finline compiler option 583
-finline-functions compiler options 584
-finline-limit compiler option 585
-finstrument-functions compiler options 586, 912
-fixed compiler option 588
-fkeep-static-consts compiler option 589, 928
-fltconsistency compiler option 590
-fma compiler option (Linux* only) 593, 880
-fmath-errno compiler option 594
-fminshared compiler option 596
-fnsplit compiler option (Linux* only) 597, 883
-fomit-frame-pointer compiler option 598, 600, 803
-fp compiler option 598, 600, 803
-fpconstant compiler option 616
-fpe-all compiler option 620
-fpe compiler option 617
-fpic compiler option 623
-fpie compiler option (Linux* only) 624
-fp-model compiler option 601, 606, 1677, 1684
  how to use 1684
-fpp compiler option 625, 889, 1232
  fpp options you can specify by using 1232
-fp-port compiler option 611, 884
-fp-relaxed compiler option 612, 885
-fpscomp all compiler option 627
-fpscomp compiler option 627
-fpscomp filesfromcmd compiler option 627
-fpscomp general compiler option 627
-fpscomp ioformat compiler option 627
-fpscomp ldio_spacing compiler option 627
-fpscomp libs compiler option 627
-fpscomp logicals compiler option 627
-fpscomp none compiler option 627
-fp-speculation compiler option 613, 886
-fp-stack-check compiler option 615, 888
-fr32 compiler option (Linux* only) 637
-FR compiler option 638
-free compiler option 638
-fsource-asm compiler option 639
-fstack-protector compiler option 640, 641, 661
-fstack-security-check compiler option 640, 641, 661
-fsyntax-only compiler option 1070
-ftrapuv compiler option 642, 1024
-ftz compiler option 643, 891, 1689
-func-groups compiler option 833
-funroll-loops compiler option 1026, 1085
-fverbose-asm compiler option 646
-fvisibility compiler option 647
-g compiler option 650, 1119, 1122
-gdwarf-2 compiler option 655
-gen-interfaces compiler option 657
-global-hoist compiler option 658, 893
-heap-arrays compiler option 662
-help compiler option 663
-i2 compiler option 691
-i4 compiler option 691
-i8 compiler option 691
-I compiler option 667
-idirafter compiler option 669
-i-dynamic compiler option 1058
-implicitnone compiler option 1099
-inline-debug-info compiler option (Linux* only) 676, 896
-inline-factor compiler option 677, 898
-inline-forceinline compiler option 679, 900
-inline-level compiler option 680, 758
-inline-max-per-compile compiler option 682, 901
-inline-max-per-routine compiler option 683, 903
-inline-max-size compiler option 685, 905
-inline-max-total-size compiler option 687, 906
-inline-min-size compiler option 688, 908
-intconstant compiler option 690
-integer-size compiler option 691
-ip compiler option 693, 914
-IPF-fltacc compiler option 698, 919
-IPF-flt-eval-method0 compiler option 696, 917
-IPF-fma compiler option 593, 880
-IPF-fp-relaxed compiler option 612, 885
-ip-no-inlining compiler option 694, 915
-ip-no-pinlining compiler option 695, 916
-ipo-c compiler option 701, 922
-ipo compiler option 699, 920, 1501
-ipo-jobs compiler option (Linux* only) 702, 923
-ipo-S compiler option 704, 925
-ipo-separate compiler option 705, 926
-i-static compiler option 1068
-isystem compiler option 706
-ivdep-parallel compiler option (Linux* only) 707, 927
-l compiler option 708
-L compiler option 709
-logo compiler option 716
-lowercase compiler option 744
-m32 compiler option 719
-m64 compiler option 719
-map-opts compiler option 721, 931

-march=pentium3 compiler option 723
-march=pentium4 compiler option 723
-march=pentium compiler option 723
-mcmodel=large compiler option (Linux* only) 724
-mcmodel=medium compiler option (Linux* only) 724
-mcmodel=small compiler option (Linux* only) 724
-m compiler option 717
-mcpu compiler option 740
-mdynamic-no-pic compiler option (Mac OS* X only) 729
-mieee-fp compiler option 590
-minstruction compiler option 730, 911
-mixed-str-len-arg compiler option 670
-mkl compiler option 732, 933
-module compiler option 734
-mp1 compiler option 737, 989
-mp compiler option 590
-mrelax compiler option (Linux* only) 738
-mtune compiler option 740
-multiple-processes compiler option 735, 743
-names as_is compiler option 744
-names lowercase compiler option 744
-names uppercase compiler option 744
-nbs compiler option 486
-no-bss-init compiler option 746, 934
-nodefaultlibs compiler option 747
-nodefine compiler option 522
-nofor-main compiler option 748
-nolib-inline compiler option 749
-nostartfiles compiler option 750
-nostdinc compiler option 1116
-nostdlib compiler option 751
-nus compiler option 486
-Ob compiler option 680, 758
-o compiler option 752
-O compiler option 753
-onetrip compiler option 764, 935
-openmp compiler option 765, 936
-openmp-lib compiler option 766, 937, 1321
-openmp-link compiler option 768, 939
-openmp-profile compiler option (Linux* only) 770, 940
-openmp-report compiler option 771, 942
-openmp-stubs compiler option 772, 943
-openmp-threadprivate compiler option 774, 944
-opt-block-factor compiler option 775, 946
-opt-jump-tables compiler option 776, 947
-opt-loadpair compiler option 778, 948
-opt-malloc-options compiler option 779
-opt-mem-bandwidth compiler option (Linux* only) 780, 949
-opt-mod-versioning compiler option 782, 951
-opt-multi-version-aggressive compiler option 783, 952
-opt-prefetch compiler option 784, 953
-opt-prefetch-initial-values compiler option 786, 955
-opt-prefetch-issue-excl-hint compiler option 787, 956
-opt-prefetch-next-iteration compiler option 788, 957
-opt-ra-region-strategy compiler option 790, 959, 1606
  example 1606
-opt-report compiler option 791, 960, 1260
-opt-report-file compiler option 793, 962
-opt-report-help compiler option 794, 963
-opt-report-phase compiler option 795, 964
-opt-report-routine compiler option 796, 965
-opt-streaming-stores compiler option 797, 966
-opt-subscript-in-range compiler option 799, 968
-Os compiler option 800
-pad compiler option 806, 971
-pad-source compiler option 807, 972
-par-affinity compiler option (Linux* only) 808, 975
-parallel compiler option 819, 986
-par-num-threads compiler option 810, 977
-par-report compiler option 811, 978
-par-schedule compiler option 814, 980
-par-threshold compiler option 818, 984
-pc compiler option 821, 987
-p compiler option 805
-P compiler option 827
-pg compiler option 805
-pie compiler option (Linux* only) 823
-prec-div compiler option 825, 990
-prec-sqrt compiler option 826, 991
-preprocess-only compiler option 827
-print-multi-lib compiler option 828
-prof-data-order compiler options 829, 992
-prof-dir compiler option 830, 994
-prof-file compiler option 832, 995
-prof-func-groups compiler option 833
-prof-func-order compiler options 834, 996
-prof-gen
  srcpos compiler option 1530, 1532, 1552
-prof-gen:srcpos compiler option
  code coverage tool 1532
  test priorization tool 1552
-prof-gen compiler option 836, 998, 1520, 1530
  related options 1520
-prof-genx compiler option 836, 998

-prof-hotness-threshold compiler option 838, 1000
-prof-src-dir compiler option 840, 1001
-prof-src-root compiler option 841, 1003
-prof-src-root-cwd compiler option 843, 1005
-prof-use compiler option 845, 1006, 1520, 1532, 1561
  code coverage tool 1532
  profmerge utility 1561
  related options 1520
-Qinstall compiler option 910
-Qlocation compiler option 929
-Qoption compiler option 969
-qp compiler option 805
-r16 compiler option 1046
-r8 compiler option 1046
-rcd compiler option 1008, 1044
-rct compiler option 1009, 1045
-real-size compiler option 1046
-recursive compiler option 1047
-reentrancy async compiler option 1049
-reentrancy none compiler option 1049
-reentrancy threaded compiler option 1049
-RTCu compiler option 511
-safe-cray-ptr compiler option 1010, 1051
-save compiler option 1012, 1053
-save-temps compiler option 1013, 1054
-scalar-rep compiler option 1015, 1056
-S compiler option 1050
-shared compiler option (Linux* only) 1057
-shared-intel compiler option 1058
-shared-libgcc compiler option 1060
-sox compiler option 1017, 1062
-stand compiler option 1063
-stand f03 compiler option 1063
-stand f90 compiler option 1063
-stand f95 compiler option 1063
-stand none compiler option 1063
-static compiler option (Linux* only) 1065
-static-intel compiler option 1068
-staticlib compiler option (Mac OS* X only) 1066
-static-libgcc compiler option 1069
-std03 compiler option 1063
-std90 compiler option 1063
-std95 compiler option 1063
-std compiler option 1063
-syntax-only compiler option 1070
-tcheck compiler option (Linux* only) 1019, 1072
-tcollect compiler option 1020, 1073
-tcollect-filter compiler option 1021, 1075
-T compiler option (Linux* only) 1071
-Tf compiler option 1061
-threads compiler option 1077
-tprofile compiler option (Linux* only) 1023, 1078
-traceback compiler option 1080
-tune pn1 compiler option 1081
-tune pn2 compiler option 1081
-tune pn3 compiler option 1081
-tune pn4 compiler option 1081
-u compiler option 1099
-U compiler option 1084
-unroll-aggressive compiler option 1027, 1086
-unroll compiler option 1026, 1085
-uppercase compiler option 744
-us compiler option 486
-use-asm compiler option 1028, 1088
-v compiler option 1089
-V compiler option 1090
-vec compiler option 1032, 1090
-vec-guard-write compiler option 1033, 1091
-vec-report compiler option 1034, 1092
-vec-threshold compiler option 1036, 1094
-vms compiler option 1095
-W0 compiler option 1099
-W1 compiler option 1099
-Wa compiler option 1098
-warn alignments compiler option 1099
-warn all compiler option 1099
-warn compiler option 1099
-warn declarations compiler option 1099
-warn errors compiler option 1099
-warn general compiler option 1099
-warn ignore_loc compiler option 1099
-warn interfaces compiler option 1099
-warn none compiler option 1099
-warn stderrors compiler option 1099
-warn truncated_source compiler option 1099
-warn uncalled compiler option 1099
-warn unused compiler option 1099
-warn usage compiler option 1099
-watch all compiler option 1105
-watch cmd compiler option 1105
-watch compiler option 1105
-watch none compiler option 1105
-watch source compiler option 1105
-WB compiler option 1106
-w compiler option 1098, 1099
-what compiler option 1107
-Wl compiler option 1110

-Wp compiler option 1111
-x compiler option 1038, 1112, 1306
-X compiler option 1116
-Xlinker compiler option 1118
-y compiler option 1070
-zero compiler option 1042, 1121
-Zp compiler option 472, 1124

5 unit specifier 197
6 unit specifier 197

A

A 2062
    edit descriptor 2062
ABORT 2319
About box
    function specifying text for 2320
ABOUTBOXQQ 2320
ABS 2321
absolute spacing function 3527
absolute value function 2321, 3509
ACCEPT 2323
ACCESS 2104, 2125, 2324
    specifier for INQUIRE 2104
    specifier for OPEN 2125
accessibility attributes
    PRIVATE 3196
    PUBLIC 3208
accessibility of modules 3196, 3208
accessing arrays efficiently 1623
access methods for files 1979
access mode function 3448
access of entities
    private 3196
    public 3208
accuracy
    and numerical data I/O 181
ACHAR 2325
ACOS 2326
ACOSD 2327
ACOSH 2328
ACTION 2104, 2125
    specifier for INQUIRE 2104
    specifier for OPEN 2125
actual arguments 511, 1920, 1947, 2650, 2927
    external procedures as 2650
    functions not allowed as 1947
actual arguments *(continued)*
    intrinsic functions as 2927
    option checking before calls 511
additional language features 2195
address
    function allocating 2990
    function returning 2409
    subroutine freeing allocated 2694
    subroutine prefetching data from 3037
adjustable arrays 1853
ADJUSTL 2328
ADJUSTR 2329
ADVANCE 1988, 3365, 3673
    specifier for READ 3365
    specifier for WRITE 3673
advanced PGO options 1530
advancing i/o 1988
advancing record I/O 238
advantages of internal procedures 261
advantages of modules 258
AIMAG 2330
AIMAX0 2995
AIMIN0 3028
AINT 2331
AJMAX0 2995
AJMIN0 3028
AKMAX0 2995
AKMIN0 3028
ALARM 2333
ALIAS 2334, 2376
    option for ATTRIBUTES directive 2376
aliases 1475, 1660
aliasing
    option specifying assumption in functions 582
    option specifying assumption in programs 573
ALIGN 2377
    option for ATTRIBUTES directive 2377
aligning data 472, 1613, 1642
    option for 472
alignment 472, 1475, 1613, 1642, 1658, 1671, 3124
    directive affecting 3124
    example 1475
    option affecting 472
    options 1671
    strategy 1475, 1671
alignment of common external data 285
ALL 2335

ALLOCATABLE 1242, 1352, 1453, 1532, 1605, 1611, 1623, 1660, 2337, 2377
  arrays as arguments 1623
  basic block 1532
  code coverage 1532
  code for OpenMP* 1352
  coding guidelines for 1611
  data flow 1453
  effects of compiler options on allocation 1660
  OpenMP* 1242
  option for ATTRIBUTES directive 2377
  performance 1352
  pipelining 1605
  visual presentation 1532
allocatable arrays 293, 1860, 1875, 1876, 1878, 2338, 2340, 2517
  allocation of 1876
  allocation status of 1876
  as dynamic objects 1875
  creating 2338
  deallocation of 1878
  freeing memory associated with 2517
  function determining status of 2340
  how to specify 1860
  mixed-language programming 293
allocatable objects
  option checking for unallocated 511
ALLOCATE 1840, 1875, 2338
  dynamic allocation 1875
  pointer assignments 1840
ALLOCATED 2340
allocating registers 1606
allocation
  of allocatable arrays 1876
  of pointer targets 1877
allocation status of allocatable arrays 1876
ALLOW_NULL 2378
  option for ATTRIBUTES directive 2378
ALOG 2979
alternate return 2191, 2429, 3394, 3586
  specifier for 2429
alternate return arguments 1928
AMAX0 2995
AMAX1 2995
AMIN0 3028
AMIN1 3028
AMOD 3040
amount of data storage
  system parameters for 438
analyzing applications 1247, 1250, 1251
  Intel(R) Debugger 1250
  Intel(R) Threading Tools 1250
AND 2821
angle brackets
  for variable format expressions 2086
ANINT 2342
ANSI character codes for Windows* Systems 2211, 2214, 2215
  chart 2215
ANY 2344
apostrophe editing 2083
APPENDMENUQQ 2345
application characteristics 1251
application performance 1251
applications
  option specifying code optimization for 753
application tests 1552
ARC 2348
ARC_W 2348
arccosine
  function returning hyperbolic 2328
  function returning in degrees 2327
  function returning in radians 2326
architectures
  coding guidelines for 1693
  option generating instructions for 480, 1081
arcs
  drawing elliptical 2348
  function testing for endpoints of 2714
arcsine
  function returning hyperbolic 2352
  function returning in degrees 2351
  function returning in radians 2350
arctangent
  function returning hyperbolic 2368
  function returning in degrees 2368
  function returning in degrees (complex) 2367
  function returning in radians 2365
  function returning in radians (complex) 2365
argument aliasing 1475, 1623, 1630, 1658
  alignment in vectorization 1475
  efficient compilation of 1658
  rules for improving I/O performance 1630
  using efficiently 1623
argument association 1920, 2183, 2185, 2186
  name 2183
  pointer 2185
  storage 2186

argument inquiry procedures
table of 2260
argument intent 2919
argument keywords 1750, 1949
BACK 1949
DIM 1949
in intrinsic procedures 1949
KIND 1949
MASK 1949
argument passing
in mixed-language programming 266, 280, 281
using %REF 3381
using %VAL 3651
argument presence function 3192
arguments 1920, 1923, 1924, 1925, 1926, 1927, 1928, 1929, 2650, 2716, 2919, 2927, 2977, 2978, 3118, 3192, 3381, 3651
actual 1920
alternate return 1928
array 1924
association of 1920
assumed-length character 1926
character constants as 1927
dummy 1920, 1924
dummy procedure 1929
function determining presence of optional 3192
function returning address of 2977, 2978
Hollerith constants as 1927
intent of 2919
optional 1923, 3118
passing by immediate value 3651
passing by reference 3381
pointer 1925
subroutine returning command-line 2716
using external and dummy procedures as 2650
using intrinsic procedures as 2927
arithmetic exception handling
/fpe options for floating-point data 1714
integer overflow 1709
arithmetic IF 2191, 2873
arithmetic shift
function performing left 2594, 3507
function performing left or right 2941
function performing right 2594, 3508
array 1694
ARRAY_VISUALIZER 2378
option for ATTRIBUTES directive 2378
array arguments 1924
array assignment 1838, 2163, 2682, 3666
masking in 2682, 3666
rules for directives that affect 2163
array association 2189
array constructors 1812, 1838
implied-DO in 1838
array declarations 1853
array descriptor
data items passing 1857, 1860, 3163
subroutine creating in memory 2668
array descriptors 295
array element order 1804
array elements 1804, 1807, 2189, 2420, 2636, 2998, 3002, 3031, 3035, 3201, 3590
association of 2189
association using EQUIVALENCE 2636
function performing binary search for 2420
function returning location of maximum 2998
function returning location of minimum 3031
function returning maximum value of 3002
function returning minimum value of 3035
function returning product of 3201
function returning sum of 3590
referencing 1804
storage of 1804
array expressions 1838
array functions
categories of 1953
for construction 3024, 3134, 3559, 3638
for inquiry 2340, 2958, 3504, 3521
for location 2998, 3031
for manipulation 2494, 2632, 3390, 3618
for reduction 2335, 2344, 2490, 3002, 3035, 3201, 3590
array operation 1694
array pointers 293, 1860
mixed-language programming 293
array procedures
table of 2265
arrays 295, 1786, 1798, 1800, 1803, 1804, 1807, 1809, 1810, 1812, 1817, 1838, 1853, 1857, 1859, 1860, 1863, 1875, 1876, 1878, 2335, 2337, 2338, 2340, 2344, 2490, 2494, 2540, 2586, 2632, 2682, 2958, 2993, 2998, 3002, 3024, 3031, 3035, 3064, 3072, 3073, 3134, 3163, 3166, 3348, 3390, 3504, 3521, 3525, 3559, 3590, 3618, 3638, 3661, 3666
adjustable 1853
allocatable 2337

arrays *(continued)*
allocation of allocatable 1876
assigning values to 1838
associating group name with 3064
as structure components 1786
as subobjects 1798
assumed-shape 1857
assumed-size 1859
as variables 1798
automatic 1853
bounds of 1800
conformable 1800
constructors 1812
creating allocatable 2338
data type of 1800
deallocation of allocatable 1878
declaring 1853, 2540
declaring using POINTER 3163
deferred-shape 1860
defining constants for 1812
determining allocation of allocatable 2340
duplicate elements in 1810
dynamic association of 1875
elements in 1804
explicit-shape 1853
extending 3390, 3559
extent of 1800
function adding a dimension to 3559
function combining 3024
function counting number of true in 2490
function determining allocation of 2340
function determining all true in 2335
function determining any true in 2344
function packing 3134
function performing circular shift of 2494
function performing dot-product multiplication of 2586
function performing end-off shift on 2632
function performing matrix multiplication on 2993
function replicating 3559
function reshaping 3390
function returning codepage in 3072
function returning language and country combinations in 3073
function returning location of maximum value in 2998
function returning location of minimum value in 3031
function returning lower bounds of 2958
arrays *(continued)*
function returning maximum value of elements in 3002
function returning minimum value of elements in 3035
function returning shape of 3504
function returning size (extent) of 3521
function returning sum of elements in 3590
function transposing rank-two 3618
function unpacking 3638
logical test element-by-element of 2682, 3666
making equivalent 1863
masked assignment of 2682, 3666
mixed-language programming 295
number of storage elements for 2540
properties of 1800
rank of 1800
referencing 1817
row-major order 295
sections of 1807
shape of 1800
size of 1800
subroutine performing quick sort on 3348
subroutine sorting one-dimensional 3525
subscript triplets in 1809
using POINTER to declare 3163, 3166
vector subscripts in 1810
volatile 3661
whole 1803
array sections 1807, 1809, 1810, 1838
assigning values to 1838
many-one 1810, 1838
subscript triplets in 1809
vector subscripts in 1810
array specifications 1853, 1857, 1859, 1860
assumed-shape 1857
assumed-size 1859
deferred-shape 1860
explicit-shape 1853
array subscripts 1804
array transposition 3618
array type declaration statements 1853
array variables 1838
ASCII character codes for Linux* and Mac OS* X Systems 2219
ASCII character codes for Windows* Systems 2211, 2212, 2213
chart 1 2212
chart 2 2213

ASCII location
    function returning character in specified position 2444
    function returning position of character in 2325
ASIN 2350
ASIND 2351
ASINH 2352
assembler
    option passing options to 1098
    option producing objects through 1028, 1088
assembler output
    generating 103
assembly files 103
assembly listing file 165, 483, 485, 646, 1050
    option compiling to 1050
    option producing with compiler comments 646
    option specifying generation of 485
    option specifying the contents of 483
ASSIGN 2352
assigned GO TO 2810
assigning values to arrays 1838
ASSIGNMENT 1942
assignments
    array 1838
    defined 1839, 2354
    derived-type 1837
    element array 2682
    generalized masked array 2682
    generic 1942
    intrinsic 1833
    intrinsic computational 2357
    masked array 3666
    masked array (generalization of) 2682
    pointer 1840
assignment statements 1833, 1834, 1836, 1838, 2163, 2354
    array 1838
    character 1836
    defining nonintrinsic 2354
    directives that affect array 2163
    logical 1836
    numeric 1834
ASSOCIATED 1840, 2360
    using to determine pointer assignment 1840
ASSOCIATEVARIABLE 2126
    specifier for OPEN 2126
association 1863, 1875, 1920, 2181, 2183, 2185, 2186, 2189, 2473, 2636
    argument 1920
association *(continued)*
    argument name 2183
    argument pointer 2185
    argument storage 2186
    array 2189
    common 2473
    dynamic 1875
    equivalence 2636
    example of 2181
    host 2183
    name 2183
    storage 2186
    types of 2181
    use 2183
    using EQUIVALENCE 1863
ASSUME_ALIGNED 2362
assumed-length character arguments 1920, 1926
assumed-length character functions 2191
assumed-shape arrays 1623, 1857
assumed-size arrays 1859
asterisk (*)
    as alternate return specifier 3586
    as assumed-length character specifier 1849, 1926
    as CHARACTER length specifier 1849, 1926
    as dummy argument 1928
    as function type length specifier 2705
    as unit specifier 197
ASYNCHRONOUS 1988, 2105, 2126, 2363
    specifier for INQUIRE 2105
    specifier for OPEN 2126
asynchronous i/o 1988, 2363
    attribute and statement denoting 2363
asynchronous I/O 253
ATAN 2365
ATAN2 2365
ATAN2D 2367
ATAND 2368
ATANH 2368
ATOMIC 1364, 2369
    using 1364
A to Z Reference 2247
ATTRIBUTES 268, 274, 280, 312, 2371, 2376, 2377, 2378, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2393, 2394
    ALIAS option 2376
    ALIGN option 2377
    ALLOCATABLE option 2377
    ALLOW_NULL option 2378
    and calling conventions 280, 312

ATTRIBUTES *(continued)*
and external naming conventions 274
ARRAY_VISUALIZER option 2378
C option 2378, 2390
DECORATE option 2381
DLLEXPORT option 2382, 2383
DLLIMPORT option 2382, 2383
EXTERN option 2384
FORCEINLINE option 2384, 2385, 2388
IGNORE_LOC option 2385
INLINE option 2384, 2385, 2388
in mixed-language programs 268
MIXED_STR_LEN_ARG option 2386, 2389
NO_ARG_CHECK option 2387
NOINLINE option 2384, 2385, 2388
NOMIXED_STR_LEN_ARG option 2386, 2389
REFERENCE option 2389, 2393
STDCALL option 2378, 2390
VALUE option 2389, 2393
VARYING option 2394
ATTRIBUTES ALIAS 2376
ATTRIBUTES ALIGN 2377
ATTRIBUTES ALLOCATABLE 2377
ATTRIBUTES ALLOW_NULL 2378
ATTRIBUTES ARRAY_VISUALIZER 2378
ATTRIBUTES C 2378, 2390
ATTRIBUTES DECORATE 2381
ATTRIBUTES DLLEXPORT 2382, 2383
ATTRIBUTES DLLIMPORT 2382, 2383
ATTRIBUTES EXTERN 2384
ATTRIBUTES FORCEINLINE 2384, 2385, 2388
attributes for data 2337, 2363, 2371, 2402, 2414, 2540, 2650, 2919, 2927, 3118, 3148, 3163, 3196, 3205, 3208, 3405, 3572, 3600, 3626, 3653, 3661
ALLOCATABLE 2337
ASYNCHRONOUS 2363
AUTOMATIC 2402
BIND 2414
declaring 3626
DIMENSION 2540
directive affecting 2371
EXTERNAL 2650
INTENT 2919
INTRINSIC 2927
OPTIONAL 3118
PARAMETER 3148
POINTER 3163
PRIVATE 3196
attributes for data *(continued)*
PROTECTED 3205
PUBLIC 3208
SAVE 3405
STATIC 3572
summary of compatible 3626
TARGET 3600
VALUE 3653
VOLATILE 3661
ATTRIBUTES IGNORE_LOC 2385
ATTRIBUTES INLINE 2384, 2385, 2388
ATTRIBUTES MIXED_STR_LEN_ARG 2386, 2389
ATTRIBUTES NO_ARG_CHECK 2387
ATTRIBUTES NOINLINE 2384, 2385, 2388
ATTRIBUTES NOMIXED_STR_LEN_ARG 2386, 2389
ATTRIBUTES REFERENCE 2389, 2393
ATTRIBUTES STDCALL 2378, 2390
ATTRIBUTES VALUE 2389, 2393
ATTRIBUTES VARYING 2394
AUTOAddArg 2395
AUTOAllocateInvokeArgs 2397
AUTODeallocateInvokeArgs 2397
AUTOGetExceptInfo 2398
AUTOGetProperty 2398
AUTOGetPropertyByID 2400
AUTOGetPropertyInvokeArgs 2400
AUTOInvoke 2401
AUTOMATIC 2402
automatic arrays 662, 1853
option putting on heap 662
automatic optimizations 1301, 1302
automation routines
table of 2314
auto-parallelization 1242, 1287, 1447, 1451, 1453
diagnostic 1287
enabling 1451
environment variables 1451
guidelines 1453
overview 1447
programming with 1453
threshold 1287
auto-parallelized loops 1287
auto-parallelizer 811, 818, 819, 978, 984, 986, 1242, 1287, 1447
controls 1242, 1287
enabling 1242
option controlling level of diagnostics for 811, 978
option enabling generation of multithreaded code 819, 986

auto-parallelizer *(continued)*
option setting threshold for loops 818, 984
AUTO routines
AUTOAddArg 2395
AUTOAllocateInvokeArgs 2397
AUTODeallocateInvokeArgs 2397
AUTOGetExceptInfo 2398
AUTOGetProperty 2398
AUTOGetPropertyByID 2400
AUTOGetPropertyInvokeArgs 2400
AUTOInvoke 2401
AUTOSetProperty 2405
AUTOSetPropertyByID 2406
AUTOSetPropertyInvokeArgs 2407
table of 2314
AUTOSetProperty 2405
AUTOSetPropertyByID 2406
AUTOSetPropertyInvokeArgs 2407
autovectorization 1696
auto-vectorization 1242, 1611
autovectorization of innermost loops 1696
auto-vectorizer 1310, 1459, 1658, 1660
allocation of stacks 1658, 1660
checking of stacks 1660
optimization for systems based on IA-32 architecture 1310
avoid
EQUIVALENCE statements 1636
inefficient data types 1636, 1696
mixed arithmetic expressions 1636, 1696
slow arithmetic operators 1636
small integer data items 1636
unnecessary operations in DO loops 1636

B

B 2045
edit descriptor 2045
BABS 2321
BACK 1949
backslash editing 2079
BACKSPACE 2407
BADDRESS 2409
BARRIER 1364, 2409
using 1364
base of model
function returning 3350
BBCLR 2825
BBITS 2826
BBSET 2827
BBTEST 2422
BDIM 2539
BEEPQQ 2410
BESJ0 2411
BESJ1 2411
BESJN 2411
Bessel functions
functions computing double-precision values of 2511
functions computing single-precision values of 2411
portability routines calculating 326
BESY0 2411
BESY1 2411
BESYN 2411
Bezier curves
functions drawing 3169, 3175
BIAND 2821
BIC 2412
BIEOR 2870
BIG_ENDIAN 2130
value for CONVERT specifier 2130
big-endian data 1658, 1664
conversion of little-endian data to 1664
big endian numeric format
porting notes 187
BINARY 2105
binary constants 1792, 2199
alternative syntax for 2199
binary direct files 208, 2136
binary editing (B) 2045
binary files 208
binary operations 1818
binary patterns
functions that shift 1951
binary raster operation constants 3498
binary sequential files 208, 2136
binary transfer of data
function performing 3616
binary values
transferring 2045
BIND 2414
BIOR 2929
BIS 2412
BIT 2416
BIT_SIZE 2416
BitBlt 3216
bit constants 1792

bit data
    model for 2227
BITEST 2422
bit fields
    function extracting 2826
    functions operating on 1951
    references to 1951
    subroutine copying 3062
bit functions 1951, 1953
    categories of 1953
bitmap file
    function displaying image from 2976
bit model 2227
bit operation procedures
    table of 2277
bit patterns
    function performing circular shift on 2945
    function performing left shift on 2943
    function performing logical shift on 2943
    function performing right shift on 2943
bit representation procedures
    table of 2277
bits
    floating-point precision bits 1711
    function arithmetically shifting left 2594, 3507
    function arithmetically shifting left or right 2941
    function arithmetically shifting right 2594, 3508
    function clearing to zero 2825
    function extracting sequences of 2826
    function logically shifting left or right 2943, 2947
    function performing exclusive OR on 2870
    function performing inclusive OR on 2929
    function performing logical AND on 2821
    function returning number of 2416
    function reversing value of 2824
    function rotating left or right 2942
    function setting to 1 2827
    function testing 2422
    model for data 2227
    precision 1711
    precision bits 1711
bitwise AND
    function performing 2821
bitwise complement
    function returning 3106
BIXOR 2870
BJTEST 2422
BKTEST 2422
BLANK 2106, 2127
    specifier for INQUIRE 2106
    specifier for OPEN 2127
blank common 2473
blank editing 2073, 2074
    BN 2073
    BZ 2074
blank interpretation 2073
blank padding 208
block constructs 1883, 2433, 2575, 2584, 2682, 2876, 3666
    CASE 2433
    DO 2575, 2584
    FORALL 2682
    IF 2876
    WHERE 3666
BLOCK DATA 2417, 2473
    and common blocks 2473
block data program units 1745, 1897, 2417, 2473
    and common blocks 2473
    effect of using DATA in 2417
    in EXTERNAL 2417
block DO 2575
    terminal statements for 2575
BLOCKSIZE 2106, 2127
    specifier for INQUIRE 2106
    specifier for OPEN 2127
BMOD 3040
BMVBITS 3062
BN 2073
BNOT 3106
bounds 511, 1800, 2958
    function returning lower 2958
    option checking 511
branching 1883, 2433, 2876
    and CASE 2433
    and IF 2876
branch specifiers 357, 1986
    END 357
    EOR 357
    ERR 357
branch statements 1883
branch target statements 1883, 1986
    in data transfer 1986
breakpoints
    use in locating source of run-time errors 359
BSEARCHQQ 2420
BSHFT 2943
BSHFTC 2945

BSIGN 3509
BTEST 2422
BUFFERCOUNT 2128
BUFFERED 2106, 2128
    specifier for INQUIRE 2106
    specifier for OPEN 2128
buffers 326, 1630
    portability routines that read and write 326
    UBC system 1630
build environment
    selecting 95
building applications
    overview 93
built-in functions 1935, 2978, 3381, 3651
    %LOC 2978
    %REF 3381
    %VAL 3651
BYTE 2424
BZ 2074

C

C 2378, 2390
    option for ATTRIBUTES directive 2378, 2390
C_ASSOCIATED 2424
C_F_POINTER 2425
C_F_PROCPOINTER 2426
C_FUNLOC 2427
C_LOC 2427
C/C++ and Fortran
    summary of programming issues 264
C/C++ interoperability 264
CABS 2321
cache
    function returning size of a level in memory 2429
    subroutine prefetching data on 3037
cache hints
    directive providing 3023
CACHESIZE 2429
cache size intrinsic 1638
CALL 2429
    using to invoke a function 2429
callback routines
    predefined QuickWin 2345, 2907, 3043
    registering for mouse events 3383
    unregistering for mouse events 3641
calling conventions
    and ATTRIBUTES directive 266
calling conventions *(continued)*
    mixed-language programming 266
    option specifying 670
calling conventions and attributes directive
    in mixed-language programs 268
calling conventions for arguments
    in mixed-language programs 280
calling C procedures from Fortran programs 312
capturing IPO output 1501
CARRIAGECONTROL 2107, 2129
    specifier for INQUIRE 2107
    specifier for OPEN 2129
carriage control 510, 2089, 2129
    option specifying for file display 510
    specifying 2129
CASE 2433
CASE DEFAULT 2433
case index 2433
case-sensitive names 108, 266, 285
CCOS 2486
CDABS 2321
CDCOS 2486
CDEXP 2647
CDFLOAT 2440
CDLOG 2979
CDSIN 3511
CDSQRT 3561
CDTAN 3598
CEILING 2441
CEXP 2647
CHANGEDIRQQ 2442
CHANGEDRIVEQQ 2443
changing number of threads 1342, 1352, 1360, 1371, 1392, 1611, 1636, 1660
    floating-point stacks 1660
    for efficiency in Intel Fortran 1636
    guidelines for Intel Architectures 1611
    in parallel region directives 1360
    in worksharing construct directives 1371
    preparing for OpenMP* programming 1352
    specifying 1342
    stacks 1660
    summary table of 1392
CHAR 2444
CHARACTER 178, 1849, 2445
    data type representation 178
    in type declaration statements 1849
CHARACTER*(*) 1849, 2191
character assignment statements 1836

Character Constant and Hollerith Arguments 1927
character constant arguments 1927
character constants 1780, 1781, 1927, 2083
  as arguments 1927
  C strings in 1781
  in format specifiers 2083
character count editing (Q) 2080
character count specifier 1989
character data
  specifying output of 2083
character data type 178, 1779, 1780, 1781, 1783, 2186, 2500
  constants 1780
  conversion rules with DATA 2500
  C strings 1781
  default kind 1779
  representation of 178
  storage 2186
  substrings 1783
character declarations 1849
character editing (A) 2062
character expressions 1823, 2961
  comparing values of 1823
  function returning length of 2961
character functions
  categories of 1953
character length
  specifying 1779
character objects
  specifying length of 1849
character operands 1823
character procedures
  table of 2275
characters
  carriage-control for printing 2129
  function returning 2444
  function returning next available 2653, 2722
  function returning position of 2820, 2829
  function writing to file 2692
  overview of Fortran 1745
  portability routines that read and write 326
character sets 1750, 2211, 2214, 2216, 3410
  ANSI 2214
  ASCII 2211
  Fortran 95/90 1750
  function scanning for characters in 3410
  Intel Fortran 1750
  key codes 2216
character storage unit 2186
character string
  function adjusting to the left 2328
  function adjusting to the right 2329
  function concatenating copies of 3390
  function locating index of last occurrence of substring in 3401
  function locating last nonblank character in 2975
  function reading from keyboard 2787
  function returning length minus trailing blanks 2962
  function returning length of 2961
  function scanning for characters in 3410
  function trimming blanks from 3619
  option affecting backslash character in 486
  subroutine sending to screen (including blanks) 3130, 3133
  subroutine sending to screen (special fonts) 3130
character string edit descriptors 2083
character string editing 2083
character strings
  as edit descriptors 2083
  comparing 3660
  function checking for all characters in 3660
  mixed-language programming 301
character substrings 1783, 1865
  making equivalent 1865
character type declaration statements 1849
character type functions 2705
character values
  transferring 2062
character variables 1779
charts for character and key codes 2211
CHDIR 2446, 3231
  POSIX version of 3231
check compiler option 359
checking
  floating-point stacks 1691
  stacks 1691
Checking the Floating-point Stack State 1691
child window
  function appending list of names to menu 3497
  function making active 3427
  function returning unit number of active 2714
  function setting properties of 3491
CHMOD 2449, 3232
  POSIX version of 3232
chunk size
  in DO directive 2579
C interoperability 306

circles
    functions drawing 2597
circular shift
    function performing 2945
clauses
    COPYIN 2485
    COPYPRIVATE 2485, 3520
    data scope attribute 2166
    DEFAULT 2521, 3139
    DEFAULT FIRSTPRIVATE 2521
    DEFAULT NONE 2521
    DEFAULT PRIVATE 2521
    DEFAULT SHARED 2521
    FIRSTPRIVATE 2579, 2657, 3139, 3418, 3520
    IF 3139
    LASTPRIVATE 2579, 2957, 3418
    NOWAIT 2579, 3418, 3520
    NUM_THREADS 3139
    ORDERED 2579
    PRIVATE 2579, 3139, 3200, 3418, 3520
    REDUCTION 2579, 3139, 3378, 3418
    SCHEDULE 2579
    SHARED 3139
CLEARSCREEN 2451
CLEARSTATUSFPQQ 2452
CLICKMENUQQ 2455
clip region
    subroutine setting 3431, 3488
CLOCK 2456
CLOCKX 2457
CLOG 2979
CLOSE 2457
CLOSE statement 234
closing files
    CLOSE statement 234
CMPLX 2459
code
    option generating for specified CPU 723
    option generating processor-specific 500, 717, 850
    option generating specialized and optimized processor-specific 1038, 1112
code coverage tool 1532
    color scheme 1532
    dynamic counters in 1532
    exporting data 1532
    syntax of 1532
code-coverage tool
    option gathering information for 836, 998
code layout 1506
codepage
    function setting current 3096
    function setting for current console 3095
    subroutine retrieving current 3082
codepage number
    function returning for console codepage 3081
    function returning for system codepage 3081
codepages
    function returning array of 3072
colon
    in array specifications 1809, 1853, 1857, 1859, 1860
colon editing 2079
color index
    function returning current 2725
    function returning for multiple pixels 2781
    function returning for pixel 2777
    function returning text 2789
    function setting current 3434
    function setting for multiple pixels 3473
    function setting for pixel 3469
color RGB value
    function returning current 2727
    function setting current 3436
COMAddObjectReference 2460
combined parallel and worksharing constructs 1359
combining arrays 3024
combining source forms 1761
COMCLSIDFromProgID 2461
COMCLSIDFromString 2461
COMCreateObjectByGUID 2462
COMCreateObjectByProgID 2463
COMGetActiveObjectByGUID 2463
COMGetActiveObjectByProgID 2464
COMGetFileObject 2465
COMInitialize 2465
COMIsEqualGUID 2468
comma
    as external field separator 2031
    using to separate input data 2066
COMMAND_ARGUMENT_COUNT 2468
command arguments
    function returning number of 2468
command interpreter
    function sending system command to 3596
command invoking a program
    subroutine returning 2730
command line
    redirecting output from 113

command line *(continued)*
running applications from 118
using the ifort command 107
using with Intel(R) Fortran 95
command-line arguments
function returning index of 2823
function returning number of 2823, 3066
subroutine returning full 2731
subroutine returning specified 2716
command-line syntax
for make and nmake command 114
command-line window
setting search paths for .mod files 258
setting search paths for include files 260
comment indicator
general rules for 1752
comment lines 1752, 1754, 1757
for fixed and tab source 1757
for free source 1754
COMMITQQ 2471
COMMON 1870, 2473
interaction with EQUIVALENCE 1870
common block association 2473
common blocks 116, 560, 877, 1870, 2417, 2473, 3124, 3207, 3405, 3425, 3661
allocating 116
defining initial values for variables in named 2417
directive modifying alignment of data in 3124
directive modifying characteristics of 3207
effect in SAVE 3405
EQUIVALENCE interaction with 1870
extending 1870
option enabling dynamic allocation of 560, 877
using derived types in 3425
volatile 3661
common external data
mixed-language programming 285
compilation
efficient 1658
optimizing 1658
compilation control statements 2157
compilation phases 99
compilation units 596, 1514
option to prevent linking as shareable object 596
compile and link
using the ifort command to 108
compiler
default actions 102
overview 81, 87
compiler *(continued)*
saving information in your executable 104
using from the command line 107
compiler directives 140, 2159, 2160, 2164, 2253, 2334, 2362, 2369, 2371, 2409, 2492, 2518, 2522, 2544, 2579, 2657, 2664, 2695, 2833, 2884, 2916, 2949, 2984, 2992, 3022, 3023, 3026, 3098, 3099, 3100, 3103, 3105, 3108, 3109, 3114, 3123, 3124, 3129, 3136, 3139, 3142, 3143, 3145, 3146, 3147, 3189, 3207, 3370, 3418, 3520, 3577, 3592, 3602, 3606, 3607, 3632, 3643, 3644, 3654, 3655, 3657, 3658, 3659, 3670
ALIAS 2334
ASSUME_ALIGNED 2362
ATOMIC 2164, 2369
ATTRIBUTES 2371
BARRIER 2164, 2409
CRITICAL 2164, 2492
DECLARE and NODECLARE 2518
DEFINE and UNDEFINE 2522, 3632
DISTRIBUTE POINT 2544
DO 2164, 2579
ENDIF 2884
FIXEDFORMLINESIZE 2657
FLUSH 2164, 2664
FREEFORM and NOFREEFORM 2695, 3098
general 2160
IDENT 2833
IF Construct 2884
IF DEFINED 2884
INTEGER 2916
IVDEP 2949
LOOP COUNT 2984
MASTER 2164, 2992
MEMORYTOUCH (i64) 3022
MEMREF_CONTROL (i64) 3023
MESSAGE 3026
OBJCOMMENT 3114
OpenMP Fortran 2164
OPTIMIZE and NOOPTIMIZE 3099, 3123
OPTIONS 3124
ORDERED 2164, 3129
overview of parallel 2164
PACK 3136
PARALLEL DO 2164, 3145
PARALLEL loop 3142, 3143
PARALLEL OpenMP Fortran 2164, 3139
PARALLEL SECTIONS 2164, 3146

compiler directives *(continued)*
PARALLEL WORKSHARE 2164, 3147
PREFETCH and NOPREFETCH 3100, 3189
prefixes for 2159
PSECT 3207
REAL 3370
rules for 2159
SECTION 2164, 3418
SECTIONS 2164, 3418
SINGLE 2164, 3520
STRICT and NOSTRICT 3103, 3577
SWP and NOSWP (i64) 3105, 3592
table of general 2253
table of OpenMP 2253
TASK 2164, 3602
TASKWAIT 2164, 3606
THREADPRIVATE 2164, 3607
UNROLL_AND_JAM and NOUNROLL_AND_JAM 3644
UNROLL and NOUNROLL 3108, 3643
VECTOR ALIGNED and VECTOR UNALIGNED 3654, 3659
VECTOR ALWAYS and NOVECTOR 3109, 3655
VECTOR NONTEMPORAL (i32, i64em) 3657, 3658
VECTOR TEMPORAL (i32, i64em) 3657, 3658
WORKSHARE 2164, 3670
compiler error conditions 331
compiler installation
option specifying root directory for 910
compiler limits 438
compiler messages 331
compiler optimization 1302
compiler optimizations 431, 1301
compiler options
affecting DOUBLE PRECISION KIND 554
affecting INTEGER KIND 691
affecting REAL KIND 1046
cross-reference tables of 1127, 1178
deprecated and removed 457
general rules for 465
how to display functional groupings 443
mapping between operating systems 138
new 444
option displaying list of 663
option mapping to equivalents 721, 931
option saving in executable or object file 1017, 1062
overview 137
overview of descriptions of 465
quick reference summary of 1127, 1178
compiler options *(continued)*
statement confirming 3122
statement overriding 3122
summary of Linux and Mac OS X options 1178
summary of Windows options 1127
compiler options used for debugging 161
compiler reports 1258, 1260, 1263, 1273, 1288, 1294, 1510
High-Level Optimization (HLO) 1273
Interprocedural Optimizations (IPO) 1263
report generation 1260
requesting with xi* tools 1510
software pipelining 1288
vectorization 1294
compiler reports quick reference 1258
compiler versions
option displaying 1107
option displaying information about 716
compile-time bounds check
option changing to warning 1106
compile-time messages
option issuing for nonstandard Fortran 1063
compiling 99, 107, 114
files from the command line 107
using makefiles 114
compiling and linking
for optimization 100
from the command line 107
mixed-language programs 311
compiling large programs 1504
compiling with IPO 1501
COMPL 2479
complementary error function
function returning 2641
COMPLEX 1774, 2478
COMPLEX(16) 1774, 1777, 3344
constants 1777
function converting to 3344
COMPLEX(4) 1774, 1775, 2459
constants 1775
function converting to 2459
COMPLEX(8) 1774, 1776, 2516
constants 1776
function converting to 2516
COMPLEX(KIND=16) representation 1731
COMPLEX(KIND=4) representation 1730
COMPLEX(KIND=8) representation 1731
COMPLEX*16 1774
COMPLEX*32 1774

COMPLEX*8 1774
complex constants
  rules for 1775
complex data
  mixed-language programming 290
complex data type 171, 290, 1730, 1731, 1774, 1775, 1776, 1777, 2186, 2459, 2516
  constants 1775, 1776, 1777
  default kind 1774
  function converting to 2459, 2516
  handling 290
  mixed-language programming 290
  native IEEE representation (COMPLEX*16) 1731
  native IEEE representation (COMPLEX*32) 1731
  native IEEE representation (COMPLEX*8) 1730
  ranges for 171
  storage 2186
complex editing 2060
complex number
  function resulting in conjugate of 2482
  function returning the imaginary part of 2330
complex operations
  option enabling algebraic expansion of 516, 855
complex values
  transferring 2049, 2060
COMPLINT 2479
COMPLLOG 2479
COMPLREAL 2479
computed GO TO 2811
COMQueryInterface 2479
COMReleaseObject 2480
COM routines
  COMAddObjectReference 2460
  COMCLSIDFromProgID 2461
  COMCLSIDFromString 2461
  COMCreateObjectByGUID 2462
  COMCreateObjectByProgID 2463
  COMGetActiveObjectByGUID 2463
  COMGetActiveObjectByProgID 2464
  COMGetFileObject 2465
  COMInitialize 2465
  COMIsEqualGUID 2468
  COMQueryInterface 2479
  COMReleaseObject 2480
  COMStringFromGUID 2481
  COMUninitialize 2482
  table of 2314
COMStringFromGUID 2481
COMUninitialize 2482
concatenation of strings
  function performing 3390
concatenation operator 1823
conditional check
  option performing in a vectorized loop 1033, 1091
conditional compilation
  directive testing value during 2522, 3632
  option defining symbol for 153, 522
  option enabling or disabling 486
conditional DO 2584
conditional parallel region execution 1242, 1287, 1360, 1371, 1512
  auto-parallelizer diagnostics 1242, 1287
  data scope attributes 1371
  inline expansion 1512
configuration files 157
  using 157
conformable arrays 1800
conformance
  to language standards 1741
CONJG 2482
conjugate
  function calculating 2482
connecting to files 3115
console
  option displaying information to 1105
console codepage
  function returning number for 3081
console keystrokes
  function checking for 3158
constant expressions 1828
constants 1763, 1766, 1771, 1772, 1773, 1775, 1776, 1777, 1779, 1780, 1812, 3148
  array 1812
  character 1780
  COMPLEX(16) 1777
  COMPLEX(4) 1775
  COMPLEX(8) 1776
  DOUBLE COMPLEX 1776
  DOUBLE PRECISION 1772
  integer 1766
  literal 1763
  logical 1779
  named 3148
  REAL(16) 1773
  REAL(4) 1771
  REAL(8) 1772
constructors
  array 1812

constructors *(continued)*
structure 1790
constructs 1883, 2433, 2575, 2584, 2682, 2876, 3666
CASE 2433
DO 2575, 2584
FORALL 2682
IF 2876
WHERE 3666
CONTAINS 1898, 1918, 2483
in internal procedures 1918
in modules and module procedures 1898
continuation indicator
general rules for 1752
continuation lines
for fixed and tab source 1757
for free source 1754
CONTINUE 2484
control 1883, 3394
returning to calling program unit 3394
control characters for printing 2089, 2129
control constructs 1883
control edit descriptors 2068, 2070, 2071, 2072, 2073, 2074, 2077, 2079, 2080
backslash 2079
BN 2073
BZ 2074
colon 2079
dollar sign 2079
for blanks 2073
forms for 2068
positional 2070
Q 2080
S 2072
Scale factor 2074
sign 2071
slash 2077
SP 2072
SS 2072
T 2070
TL 2071
TR 2071
X 2071
controlling expression
using to evaluate block of statements 2433
control list 1981
control-list specifiers 1981, 1983, 1984, 1985, 1986, 1988, 1989
defining variable for character count 1989
for advancing or nonadvancing i/o 1988
control-list specifiers *(continued)*
for asynchronous i/o 1988
for transfer of control 1986
identifying the i/o status 1985
identifying the record number 1985
identifying the unit 1983
indicating the format 1984
indicating the namelist group 1985
control procedures
table of 2258
control statements 1883, 2258
table of 2258
control transfer 1883, 1885, 2429, 2433, 2603, 2810, 2811, 2813, 2873, 2875, 2876, 3394
with arithmetic if 2873
with branch statements 1883
with CALL 2429
with CASE 2433
with DO 1885
with END 2603
with GO TO 2810, 2811, 2813
with IF construct 2876
with logical IF 2875
with RETURN 3394
control variables
function setting value of dialog 2562
control word
floating-point 1710
setting and retrieving floating-point 1706
subroutines returning floating-point 2732, 3415
subroutines setting floating-point 2959, 3438
conventions
in the documentation 81
conversion
double-precision to single-precision type 3371
effect of data magnitude on G format 2058
from integers to RGB color value 3399
from RGB color value to component values 2917
function performing logical 2983
function resulting in COMPLEX(16) type 3344
function resulting in complex type 2459
function resulting in double-complex type 2516
function resulting in integer type 2910
function resulting in quad-precision type 3345, 3346, 3347, 3348
function resulting in real type 3371, 3402
function resulting in single-precision type 2887, 3371

conversion *(continued)*
functions resulting in double-precision type 2513, 2536, 2537, 2574, 2593, 2833
INTEGER(2) to INTEGER(4) 2983
INTEGER(4) to INTEGER(2) 3508
record structures to derived types 2201
to nearest integer 2441, 2663
to truncated integer 2910
conversion rules for numeric assignment 1834
CONVERT 181, 194, 195, 196, 2107, 2130
specifier for INQUIRE 2107
specifier for OPEN 181, 194, 195, 196, 2130
coordinates
subroutine converting from physical to viewport 2798
subroutine converting from viewport to physical 2775
subroutine returning for current graphics position 2735
COPYIN 1342, 2485, 3139, 3145, 3146, 3607
for THREADPRIVATE common blocks 3607
in PARALLEL directive 3139
in PARALLEL DO directive 3145
in PARALLEL SECTIONS directive 3146
summary of data scope attribute clauses 1342
COPYPRIVATE 2485, 3520
in SINGLE directive 3520
correct usage of countable loop 1466
COS 1466, 2486
correct usage of 1466
COSD 2487
COSH 2488
cosine
function returning 2486, 2487
function returning hyperbolic 2488
function with argument in degrees 2487
function with argument in radians 2486
COTAN 2488
COTAND 2489
cotangent
function returning 2488, 2489
function with argument in degrees 2489
function with argument in radians 2488
COUNT 2490
counters for dynamic profile 1579
country
function setting current 3096
subroutine retrieving current 3082
CPU
option generating code for specified 723
option performing optimizations for specified 740
CPU_TIME 2492
CPU dispatch
automatic 1310
CPU time 1511, 1552, 1611, 1643, 2492, 2515, 2595, 2643
DPI lists 1552
for inline function expansion 1511
function returning elapsed 2515, 2595, 2643
multithreaded applications 1611
CQABS 2321
CQCOS 2486
CQEXP 2647
CQLOG 2979
CQSIN 3511
CQSQRT 3561
CQTAN 3598
CRAY 2130
value for CONVERT specifier 2130
CreateFile
creating a jacket to 2148
create libraries using IPO 1508
CreateProcess 1495
CreateThread 1481
CRITICAL 2492
critical errors
subroutine controlling prompt for 3444
cross-iteration dependencies 1352
cross reference
of Linux and Mac OS X options 1178
of Windows options 1127
Quick Reference Guide and Cross Reference 1127, 1178
C run-time exceptions 426, 2751
function returning pointer to 2751
CSHIFT 2494
CSIN 3511
CSMG 2497
CSQRT 3561
C strings 1781
C-style escape sequence 1781
CTAN 3598
CTIME 2497
C-type character string 1781
currency string
function returning for current locale 3074

current date
subroutines returning 2506, 2507, 2508, 2509, 2738, 2830, 2832
current locale
function returning information about 3083
cursor
function controlling display of 2543
function setting the shape of 3466
CYCLE 2498

D

D 2052
edit descriptor 2052
DABS 2321
DACOS 2326
DACOSD 2327
DACOSH 2328
DASIN 2350
DASIND 2351
DASINH 2352
data
compiler option affecting 116
locating unaligned 164
DATA 2500
data alignment 1642
data conversion rules
for numeric assignment 1834
data edit descriptors 2039, 2040, 2042, 2044, 2045, 2046, 2048, 2050, 2052, 2054, 2056, 2058, 2061, 2062, 2065
A 2062
B 2045
D 2052
default widths for 2065
E 2052
EN 2054
ES 2056
F 2050
forms for 2040
G 2058
I 2044
L 2061
O 2046
rules for numeric 2042
Z 2048
data editing
specifying format for 1984
data file
converting unformatted files 210
handling I/O errors 353
limitations in converting unformatted files 1721
RECL units for unformatted files 187
dataflow analysis 1242, 1447
data format
alignment 1475, 1671
alignment of common external 285
allocatable arrays in mixed-language programming 293
array pointers in mixed-language programming 293
arrays in mixed-language programming 295
big endian unformatted file formats 181
character strings in mixed-language programming 301
common external in mixed-language programming 285
dependence 1287, 1605
exchanging and accessing in mixed-language programming 280, 289
floating-point 1724
formats for unformatted files 181
little endian unformatted file formats 181
methods of specifying 188
mixed-language programming 289
nonnative numeric formats 181
options 1660
partitioning 1453
passing as arguments in mixed-language programming 266, 280
pointers in mixed-language programming 294
porting non-native data 187
prefetching 1581, 1591
scope attribute clauses 1342
sharing 1242
statement controlling 2685
strings in mixed-language programming 301
structure 1475
structures in mixed-language programming 305
type 1242, 1459, 1636
user-defined types in mixed-language programming 305
VAX* floating-point formats 181
data initialization 2500
DATAN 2365
DATAN2 2365
DATAN2D 2367

DATAND 2368
DATANH 2368
data objects 1763, 2186, 2371, 2500, 2609, 2636, 3064, 3163, 3373, 3405, 3579, 3626, 3661
  assigning initial values to 2500
  associating with group name 3064
  association of 2186
  declaring type of 3626
  directive specifying properties of 2371
  record structure 2609, 3373, 3579
  retaining properties of 3405
  specifying pointer 3163
  storage association of 2636
  unpredictable values of 3661
data ordering optimization 1566
data prefetches 1602
data representation 432, 2223, 2224, 2225, 2227
  and portability considerations 432
  model for bit 2227
  model for integer 2224
  model for real 2225
data representation models 2223
  intrinsic functions providing data for 2223
data scope attribute clauses 1342, 2166
data storage
  and portability considerations 432
  argument passing in mixed-language programming 266, 280
  association 2186
  common external in mixed-language programming 285
  mixed-language programming 289
data transfer 1979, 2077, 2136, 3616
  function for binary 3616
  indicating end of 2077
  specifying mode of 2136
data transfer statements 1979, 1980, 1981, 1983, 1984, 1985, 1986, 1988, 1989, 1990, 1991, 1995, 2323, 3194, 3365, 3398, 3673
  ACCEPT 2323
  ADVANCE specifier in 1988
  ASYNCHRONOUS specifier in 1988
  components of 1980
  control list in 1981
  control specifiers in 1981
  END specifier in 1986
  EOR specifier in 1986
  ERR specifier in 1986
  FMT specifier in 1984
data transfer statements *(continued)*
  i/o lists in 1990
  implied-do lists in 1995
  input 2323, 3365
  IOSTAT specifier in 1985
  list items in 1991
  NML specifier in 1985
  output 3194, 3398, 3673
  PRINT 3194
  READ 3365
  REC specifier in 1985
  REWRITE 3398
  SIZE specifier in 1989
  UNIT specifier in 1983
  WRITE 3673
data type
  declarations 3626
  explicit 1799
  implicit 1800
  specifying for variables 1799
data types 171, 174, 176, 178, 179, 181, 188, 266, 280, 289, 293, 294, 295, 301, 305, 1613, 1698, 1728, 1729, 1730, 1731, 1763, 1765, 1769, 1774, 1778, 1779, 1784, 1799, 1800, 1821, 1847, 2186, 2424, 2445, 2478, 2530, 2587, 2588, 2617, 2889, 2915, 2982, 3368, 3620, 3626
  allocatable arrays in mixed-language programming 293
  array pointers in mixed-language programming 293
  arrays in mixed-language programming 295
  big endian unformatted file formats 181
  BYTE 2424
  CHARACTER 1779, 2445
  character representation 178
  character strings in mixed-language programming 301
  COMPLEX 1728, 1774, 2478
  declaring 3626
  derived 1784, 2530, 2617, 3620
  DOUBLE COMPLEX 1728, 1774, 2587
  DOUBLE PRECISION 1728, 1769, 2588
  efficiency 1698
  explicit 1799
  formats for unformatted files 181
  Hollerith representation 179
  IEEE S_float representation (COMPLEX*8) 1730
  IEEE S_float representation (REAL*4) 1728

data types *(continued)*
IEEE T_float representation (COMPLEX*16) 1731
IEEE T_float representation (COMPLEX*32) 1731
IEEE T_float representation (REAL*8) 1729
implicit 1800
INTEGER 174, 1765, 2915
intrinsic 1763
little endian unformatted file formats 181
LOGICAL 176, 1778, 2982
methods of using nonnative formats 188
mixed-language programming 289
native data representation 171
native IEEE* floating-point representation 1728
noncharacter 1847
obtaining unformatted numeric formats 188
of scalar variables 1799
passing as arguments in mixed-language programming 266, 280
pointers in mixed-language programming 294
ranges for denormalized native floating-point data 171
ranges for native numeric types 171
ranking in expressions 1821
REAL 1728, 1769, 3368
statement overriding default for names 2889
storage for 171
storage requirements for 2186
strings in mixed-language programming 301
structures in mixed-language programming 305
user-defined
in mixed-language programming 305
DATE 2506, 2507, 2508, 2509, 2738, 2830, 2832, 2952, 2953, 3076, 3441, 3640
function returning for current locale 3076
function returning Julian 2952, 2953
function setting 3441
routine to prevent Year 2000 problem 2509
subroutines returning 2508, 2509, 2738, 2830, 2832
subroutines returning current system 2506, 2507, 2508, 2509
subroutine unpacking a packed 3640
DATE_AND_TIME 2509
DATE4 2508
date and time
routine returning as ASCII string 2652
subroutine packing values for 3138
subroutine returning 4-digit year 2509
subroutine returning current system 2509
date and time format
for NLS functions 3083
date and time routines
table of 2263
DAZ flag 1689
DBESJ0 2511
DBESJ1 2511
DBESJN 2511
DBESY0 2511
DBESY1 2511
DBESYN 2511
DBLE 2513
DCLOCK 2515
DCMPLX 2516
DCONJG 2482
DCOS 2486
DCOSD 2487
DCOSH 2488
DCOTAN 2488
DCOTAND 2489
DDIM 2539
deadlocks 1491
DEALLOCATE 2517
debugger
Intel(R) Debugger (IDB) 168
limitations 168
multithread programs 168
use in locating run-time error source 359
debugging 110, 161, 168, 526, 529, 3026
directive specifying string for 3026
executables 110
multithread programs 168
option affecting information generated 526, 529
option specifying settings to enhance 526, 529
preparing Fortran programs for debugging 161
debugging Fortran programs 161
debugging statement indicator
for fixed and tab source 1757
for free source 1754
debug information
option generating for PARAMETERs used 532
option generating full 650, 1119, 1122
option requesting Visual C++ compatible 1030
option saving to program database file 822
debug library
option searching for unresolved references in 524
debug statements
option compiling 523, 856

decimal exponents
    function returning range of 3363
decimal precision
    function returning 3189
declarations 1845, 2250, 2623, 3634
    MAP 2623, 3634
    table of procedures for data 2250
    UNION 2623, 3634
declaration statements 1845, 1847, 1849, 1852, 1853
    for arrays 1853
    for character types 1849
    for derived types 1852
    for noncharacter types 1847
DECLARE 140, 2518
    equivalent compiler option for 140
DECODE 2519
DECORATE 2381
    option for ATTRIBUTES directive 2381
decorated name 268, 276
DEFAULT 1342, 1345, 2382, 2521, 3139, 3145, 3146
    in PARALLEL directive 3139
    in PARALLEL DO directive 3145
    in PARALLEL SECTIONS directive 3146
    option for ATTRIBUTES directive 2382
    summary of data scope attribute clauses 1342
    using 1345
default actions of the compiler 102
DEFAULTFILE 2133
default file name 2134
DEFAULT FIRSTPRIVATE 2521
default initialization 1785
default libraries 97
DEFAULT NONE 2521
default pathnames 227
DEFAULT PRIVATE 2521
DEFAULT SHARED 2521
default tools 97
default widths for data edit descriptors 2065
deferred-shape arrays 1623, 1860
DEFINE 140, 153, 2522, 3632
    equivalent compiler option for 140
    using to detect preprocessor symbols 153
defined assignment 1839, 2354
defined operations 1827, 1940
defined variables 1798
DEFINE FILE 2523
defining generic assignment 1942
defining generic operators 1940
DELDIRQQ 2525
DELETE
    alternative syntax for statement 2200
DELETEMENUQQ 2527
DELETE value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
DELFILESQQ 2528
DELIM 2108, 2133
    specifier for INQUIRE 2108
    specifier for OPEN 2133
denormal exceptions 1611, 1694
denormalized numbers (IEEE*) 171, 1722, 1728
    double-precision range 171
    exponent value of 1728
    NaN values 1722
    S_float range 171
    single-precision range 171
    T_float range 171
denormal numbers 1611, 1688
    flush-to-zero 1611
denormal results
    option flushing to zero 643, 891
denormals 1688
denormals-are-zero 1611
dependence analysis
    directive assisting 2949
deprecated compiler options 457
dequeuing 1364
DERF 2640
DERFC 2641
derived data types 1784
derived-type assignments 1837
derived-type components 1623, 1784, 1785, 1786
    default initialization of 1785
    referencing 1786
derived-type data
    components of 1784, 1785
    definition of 1785
derived-type declaration statements 1852
derived-type definition 1784, 1785, 3425
    preserving the storage order of 3425
derived-type items
    directive specifying starting address of 3136
derived types 305, 1784, 1786, 1790, 1817, 1837, 2530, 2617, 3425, 3620
    assignments with 1837
    components of 1786
    declaring 2530, 2617, 3620
    equivalencing 3425
    mixed-language programming 305

derived types *(continued)*
referencing 1817
specifying scalar values of 1790
storage for 1784
using in common blocks 3425
derived type statement 2530, 2617, 3620
determining parallelization 1242
development environment
choosing 95
devenv command 129
devices
associating with units 3115
logical 197
devices and files 197
device-specific blocksize 1630
DEXP 2647
DFLOAT 2536
DFLOATI 2537
DFLOATJ 2537
DFLOATK 2537
DFLOTI 2536
DFLOTJ 2536
DFLOTK 2536
diag compiler option 331
diagnostic messages
option affecting which are issued 533, 539, 857, 863, 1099
option controlling auto-parallelizer 533, 539, 811, 857, 863, 978
option controlling display of 533, 539, 857, 863
option controlling OpenMP 533, 539, 857, 863
option controlling OpenMP parallelizer 771, 942
option controlling source control 533, 539, 857, 863
option controlling vectorizer 533, 539, 857, 863, 1034, 1092
option displaying ID number values of 551, 875
option enabling or disabling 533, 539, 857, 863
option enabling parallel lint 545, 869
option issuing only once 552, 876
option printing enabled 538, 862
option processing include files and source files for 544, 867
option sending to file 548, 872
option stopping compilation after printing 538, 862
diagnostic reports 1287
diagnostics 336, 1242, 1287, 1461
auto-parallelizer 1242, 1287
dialog boxes
assigning event handlers to controls in 2565
deallocating memory associated with 2573
displaying modeless 2557
function assigning event handlers to controls 2565
functions displaying 2555
functions initializing 2551
message for modeless 2552
subroutine closing 2546
subroutine setting title of 2572
subroutine updating the display of 2547
dialog box messages
subroutine setting 3464
dialog control boxes
function sending a message to 2561
dialog control variable
functions retrieving state of 2549
functions setting value of 2562
dialog routines 2312, 2546, 2547, 2549, 2551, 2552, 2555, 2557, 2561, 2562, 2565, 2567, 2568, 2572, 2573
DLGEXIT 2546
DLGFLUSH 2547
DLGGET 2549
DLGGETCHAR 2549
DLGGETINT 2549
DLGGETLOG 2549
DLGINIT 2551
DLGINITWITHRESOURCEHANDLE 2551
DLGISDLGMESSAGE 2552
DLGISDLGMESSAGEWITHDLG 2552
DLGMODAL 2555, 2567
DLGMODALWITHPARENT 2555
DLGMODELESS 2557
DLGSENDCTRLMESSAGE 2561
DLGSET 2562
DLGSETCHAR 2562
DLGSETCTRLEVENTHANDLER 2565
DLGSETINT 2562
DLGSETLOG 2562
DLGSETRETURN 2567
DLGSETSUB 2568
DLGSETTITLE 2572
DLGUNINIT 2573
table of 2312
difference operators 1379
differential coverage 1532
DIGITS 2538
DIM 2539

DIMAG 2330
DIMENSION 2540
dimensions 1800, 2958
    function returning lower bounds of 2958
DINT 2331
DIRECT 2109
direct-access files 208, 246
    RECL values 246
direct access mode 1979
direct-access READ statements 2012, 2013
    rules for formatted 2013
    rules for unformatted 2013
direct-access WRITE statements 2026, 2027
    rules for formatted 2026
    rules for unformatted 2027
direct file access 210
direction keys
    function determining behavior of 3150
directive prefixes 2159
directives 140, 146, 1596, 1613, 2159, 2160, 2164, 2334, 2362, 2369, 2371, 2409, 2492, 2518, 2522, 2544, 2579, 2657, 2664, 2695, 2833, 2884, 2916, 2949, 2984, 2992, 3022, 3023, 3026, 3098, 3099, 3100, 3103, 3105, 3108, 3109, 3114, 3123, 3124, 3129, 3136, 3139, 3142, 3143, 3145, 3146, 3147, 3189, 3207, 3370, 3418, 3520, 3577, 3592, 3602, 3606, 3607, 3632, 3643, 3644, 3654, 3655, 3657, 3658, 3659, 3670
    ALIAS 2334
    ASSUME_ALIGNED 2362
    ATOMIC 2164, 2369
    ATTRIBUTES 2371
    BARRIER 2164, 2409
    commons 1613
    compiler 140
    CRITICAL 2164, 2492
    dcommons 1613
    DECLARE 2518
    DEFINE 2522, 3632
    DISTRIBUTE POINT 2544
    DO 2579
    FIXEDFORMLINESIZE 2657
    FLUSH 2664
    fpp 146
    FREEFORM 2695, 3098
    general 2160
    IDENT 2833
    IF 2884
directives *(continued)*
    IF DEFINED 2884
    INTEGER 2916
    IVDEP 1596, 2949
    LOOP COUNT 2984
    MASTER 2164, 2992
    MEMORYTOUCH (i64) 3022
    MEMREF_CONTROL (i64) 3023
    MESSAGE 3026
    NODECLARE 2518
    NOFREEFORM 2695, 3098
    NOOPTIMIZE 3099, 3123
    NOPARALLEL loop 3142, 3143
    NOPREFETCH 3100, 3189
    NOSTRICT 3103, 3577
    NOSWP (i64) 3105, 3592
    NOUNROLL 3108, 3643
    NOUNROLL_AND_JAM 3644
    NOVECTOR 1596, 3109, 3655
    OBJCOMMENT 3114
    OPTIMIZE 3099, 3123
    OPTIONS 3124
    ORDERED 2164, 3129
    overview of parallel 2164
    PACK 3136
    PARALLEL DO 2164, 3145
    PARALLEL loop 3142, 3143
    PARALLEL OpenMP Fortran 2164, 3139
    PARALLEL SECTIONS 2164, 3146
    PARALLEL WORKSHARE 2164, 3147
    PREFETCH 3100, 3189
    prefixes for 2159
    PSECT 3207
    REAL 3370
    records 1613
    SECTION 2164, 3418
    SECTIONS 2164, 3418
    sequence 1613
    SINGLE 2164, 3520
    STRICT 3103, 3577
    structure 1613
    SWP (i64) 3105, 3592
    syntax rules for 2159
    TASK 2164, 3602
    TASKWAIT 2164, 3606
    THREADPRIVATE 2164, 3607
    UNDEFINE 2522, 3632
    UNROLL 3108, 3643
    UNROLL_AND_JAM 3644

directives *(continued)*
VECTOR 1596
VECTOR ALIGNED 3654, 3659
VECTOR ALWAYS 1596, 3109, 3655
VECTOR NONTEMPORAL 1596
VECTOR NONTEMPORAL (i32, i64em) 3657, 3658
VECTOR TEMPORAL (i32, i64em) 3657, 3658
VECTOR UNALIGNED 3654, 3659
WORKSHARE 2164, 3670
directives for OpenMP* 1359, 1360, 1364, 1371
ATOMIC 1364
BARRIER 1364
CRITICAL 1364
DO 1371
END DO 1371
END PARALLEL 1360
END PARALLEL DO 1359
END PARALLEL SECTIONS 1359
END SECTIONS 1371
END SINGLE 1371
FLUSH 1364
MASTER 1364
ORDERED 1364
PARALLEL 1360
PARALLEL DO 1359
PARALLEL SECTIONS 1359
PARALLEL WORKSHARE 1359
SECTION 1371
SECTIONS 1371
SINGLE 1371
WORKSHARE 1359
directory
function changing the default 2446
function creating 2989
function deleting 2525
function returning full path of 2703
function returning path of current working 2737
function specifying current as default 2442
inquiring about properties of 2903
option adding to start of include path 706
option specifying for executables 503
option specifying for includes and libraries 503
directory path
function splitting into components 3528
directory procedures
table of 2288
disabling
efficient use of 1630
function splitting 1520
disabling *(continued)*
inlining 1512
disabling optimization 1314
disassociated pointer 3110, 3112
function returning 3110
DISPLAYCURSOR 2543
DISPOSE 2134
specifier for OPEN 2134
DISPOSE specifier for CLOSE 2457
DISP specifier for CLOSE 2457
DISTRIBUTE POINT 1591, 2544
using 1591
division expansion 431
DLGEXIT 2546
DLGFLUSH 2547
DLGGET 2549
DLGGETCHAR 2549
DLGGETINT 2549
DLGGETLOG 2549
DLGINIT 2551
DLGINITWITHRESOURCEHANDLE 2551
DLGISDLGMESSAGE 2552
DLGISDLGMESSAGEWITHDLG 2552
DLGMODAL 2555
DLGMODALWITHPARENT 2555
DLGMODELESS 2557
DLGSENDCTRLMESSAGE 2561
DLGSET 2562
DLGSETCHAR 2562
DLGSETCTRLEVENTHANDLER 2565
DLGSETINT 2562
DLGSETLOG 2562
DLGSETRETURN 2567
DLGSETSUB 2568
DLGSETTITLE 2572
DLGUNINIT 2573
DLLEXPORT 2382, 2383
option for ATTRIBUTES directive 2382, 2383
DLLIMPORT 2382, 2383
option for ATTRIBUTES directive 2382, 2383
dllimport functions
option controlling inlining of 897
DLOG 2979
DLOG10 2980
DMAX1 2995
DMIN1 3028
DMOD 3040
DNINT 2342
DNUM 2574

DO 1886, 2162, 2575, 2579, 2584
    block 1886
    directive 2579
    iteration 1886, 2575
    loop control 1886
    nonblock 1886
    rules for directives that affect 2162
    WHILE 2584
DO constructs 1398, 1466, 1611, 1623, 1636, 1886, 1888, 1891, 2484, 2498, 2575, 2584, 2645
    execution of 1886
    extended range of 1891
    forms for 1886
    immediate termination of 2645
    interrupting 2498
    nested 1888
    numbers 1398, 1611, 1636
    order of 1623
    termination statement for labeled 2484
    WHILE 2584
Documentation
    conventions for 81
    platform labels in 81
dollar sign ()
    in names 1748
dollar sign editing 2079
DO loop iterations
    option specifying scheduling algorithm for 814, 980
DO loops
    directive assisting dependence analysis of 2949
    directive controlling unrolling and jamming 3644
    directive enabling non-streaming stores 3657, 3658
    directive enabling prefetching of arrays in 3100, 3189
    directive enabling software pipelining for 3105, 3592
    directive enabling streaming stores 3657, 3658
    directive enabling vectorization of 3109, 3655
    directive facilitating auto-parallelization for 3142, 3143
    directive specifying alignment of data in 3654, 3659
    directive specifying distribution for 2544
    directive specifying the count for 2984
    directive specifying the unroll count for 3108, 3643
    enabling jamming 3644
    limiting loop unrolling 3108, 3643
    option executing at least once 764, 935
    rules for directives that affect 2162
    statement terminating 2604
DO loops *(continued)*
    statement to skip iteration of 2498
    statement transferring control from 2645
    terminal statement for 2484
DOT_PRODUCT 2586
dot-product multiplication
    function performing 2586
double colon separator 3626
DOUBLE COMPLEX 554, 1776, 2516, 2587
    constants 1776
    function converting to 2516
    option specifying default KIND for 554
DOUBLE PRECISION 554, 1772, 2513, 2536, 2537, 2574, 2588, 2593, 2833
    constants 1772
    functions converting to 2513, 2536, 2537, 2574, 2593, 2833
    option specifying default KIND for 554
double-precision product
    function producing 2589
double-precision real 1769
double-precision real editing (D) 2052
DO WHILE 2584
DO WHILE loops 2584, 2604, 2645
    statement terminating 2604
    statement transferring control from 2645
DPROD 2589
DRAND 2590
DRANDM 2590
DRANSET 2592
DREAL 2593
drive
    function returning available space on 2741
    function returning path of 2740
    function returning total size of 2741
    function specifying current as default 2443
drive procedures
    table of 2288
driver tool commands
    option specifying to show and execute 1089
    option specifying to show but not execute 556
drives
    function returning available 2744
DSHIFTL 2594
DSHIFTR 2594
DSIGN 3509
DSIN 3511
DSIND 3512
DSINH 3513

DSQRT 3561
DTAN 3598
DTAND 3599
DTANH 3599
DTIME 2595
dual-core 1242
dual core thread affinity 1418
dummy arguments 1623, 1630, 1785, 1857, 1859, 1920, 1929, 2919, 3118, 3653
  default initialization of derived-type 1785
  optional 3118
  specifying argument association for 3653
  specifying intended use of 2919
  specifying intent for 2919
  taking shape from an array 1857
  taking size from an array 1859
dummy procedure arguments 1929
dummy procedures 1897, 1929, 2650
  interfaces for 1929
  using as actual arguments 2650
dumping profile information 1576, 1578
DYLD_LIBRARY_PATH environment variable 129
dynamic allocation 1875
dynamic association 1875
dynamic information 1398, 1519, 1530, 1574, 1576, 1579
  dumping profile information 1576
  files 1530
  resetting profile counters 1579
  threads 1398
dynamic-information files 1520
dynamic libraries
  option invoking tool to generate 559
dynamic linker
  option specifying an alternate 558
dynamic-linking of libraries
  option enabling 504
dynamic-link libraries (DLLs)
  option searching for unresolved references in 726, 728
  option specifying the name of 563
dynamic memory allocation 1875
dynamic objects 1875
dynamic shared object
  option producing a 1057
dyn files 1520, 1530, 1574, 1576, 1579

E

E 2052
  edit descriptor 2052
ebp register
  option determining use in optimizations 598, 600, 803
edit descriptor 2079
edit descriptors 2031, 2039, 2044, 2045, 2046, 2048, 2050, 2052, 2054, 2056, 2058, 2061, 2062, 2068, 2070, 2071, 2072, 2073, 2074, 2077, 2079, 2080, 2083, 2084, 2086, 2685
  A 2062
  apostrophe 2083
  B 2045
  backslash 2079
  BN 2073
  BZ 2074
  character string 2083
  colon 2079
  control 2068
  D 2052
  data 2039
  dollar sign 2079
  E 2052
  EN 2054
  ES 2056
  F 2050
  for interpretation of blanks 2073
  G 2058
  H 2084
  Hollerith 2084
  I 2044
  L 2061
  O 2046
  P 2074
  Q 2080
  quotation mark 2083
  repeatable 2039
  repeat specifications for 2086
  S 2072
  scale factor 2074
  slash 2077
  SP 2072
  SS 2072
  summary 2031
  T 2070
  TL 2071
  TR 2071

edit descriptors *(continued)*
X 2071
Z 2048
edit lists 2031
efficiency 1696
efficient 1242, 1512, 1520, 1623, 1630, 1636, 1658
auto-parallelizer 1242
compilation 1658
implied-DO loop collapsing 1630
inlining 1512
parallelizer 1242
PGO options 1520
use of arrays 1623
use of record buffers 1630
efficient data types 1698
ELEMENTAL 2596, 2705, 3586
in functions 2705
in subroutines 3586
elemental intrinsic procedures 1934, 1947
references to 1934
elemental user-defined procedures 2596
element array assignment 2682
elements
function returning number of 3521
ELLIPSE 2597
ELLIPSE_W 2597
ellipses
functions drawing 2597
elliptical arcs
drawing 2348
ELSE WHERE 2600
EN 2054
ENCODE 2601
END 357, 1986, 2603, 3365, 3405
retaining data after execution of 3405
specifier for READ 3365
using the specifier 357, 1986
END DO 2604
ENDFILE 2605
endian
big and little types 181
endian data 1242, 1342, 1360, 1379, 1383, 1398, 1402, 1451, 1466, 1475, 1530, 1574, 1576, 1578, 1611, 1630, 1664
and OpenMP* extension routines 1402
auto-parallelization 1451
denormal 1611
dumping profile information 1576
for auto-parallelization 1451
endian data *(continued)*
for little endian conversion 1664
for profile-guided optimization 1574
FORT_BUFFERED 1630
loop constructs 1466
OMP_NUM_THREADS 1360
OMP_SCHEDULE 1342
OpenMP* 1383
parallel program development 1242
PROF_DIR 1574
PROF_DUMP_INTERVAL 1578
routines overriding 1398
using OpenMP* 1379
using profile-guided optimization 1530
vectorization 1475
ENDIF directive 2884
end-of-file condition
function checking for 2629
intrinsic checking for 2939
end-of-file record
function checking for 2629
retaining data after execution of 3405
statement writing 2605
end-off shift on arrays
function performing 2632
end-of-record condition
i/o specifier for 1986
intrinsic checking for 2940
END PARALLEL DO 1359
using 1359
END WHERE 2626
engineering-notation editing (EN) 2054
enhancing optimization 1251
enhancing performance 1251
entities
private 3196
public 3208
ENTRY 1944, 1945, 2627
in functions 1944
in subroutines 1945
entry points 1944, 1945, 2627
for function subprograms 1944
for subroutine subprograms 1945
environment variables
compile-time 129
converting nonnative numeric data 190
F_UFMTENDIAN 192
FOR_GENERATE_DEBUG_EXCEPTION 132
FORT_CONVERT_ext 132, 189

environment variables *(continued)*
FORT_CONVERT.ext 132, 189
FORT_CONVERTn 132, 190
FORTn 132
function adding new 3442
function returning value of 2748
function scanning for 3412
function setting value of 3442
run-time 132
setting 129
setting with ifortvars file 127
subroutine getting the value of 2745
used with traceback information 412
EOF 2629
EOR 357, 1986, 3365
specifier for READ 3365
using the specifier 357
EOSHIFT 2632
EPSILON 2635
EQUIVALENCE 1636, 1863, 1865, 1870, 2636
effect on run-time efficiency 1636
interaction with COMMON 1870
using with arrays 1863
using with substrings 1865
equivalence association 2636
equivalent arrays
making 1863
equivalent substrings
making 1865
ERF 2640
ERFC 2641
ERR 357, 1986, 3365, 3673
specifier for READ 3365
specifier for WRITE 3673
using the specifier 357
errno names 2871
error
subroutine sending last detected to standard error stream 3159
error codes 361, 2871
error conditions
i/o specifier for 1986
subroutine returning information on 2642
error functions
functions returning 2640, 2641
error handling 353, 357, 358
overriding default 357
processing performed by Intel(R) Fortran RTL 353
supplementing default 357
error handling *(continued)*
user controls in I/O statements 358
error handling procedures
table of 2288
error messages 361
error numbers 2871
errors
during build process 331
FPCW$24 1710
FPCW$53 1710
FPCW$64 1710
FPCW$AFFINE 1710
FPCW$CHOP 1710
FPCW$DENORMAL 1710
FPCW$DOWN 1710
FPCW$INEXACT 1710
FPCW$INVALID 1710
FPCW$MCW_EM 1710
FPCW$MCW_IC 1710
FPCW$MCW_PC 1710
FPCW$MCW_RC 1710
FPCW$NEAR 1710
FPCW$OVERFLOW 1710
FPCW$PROJECTIVE 1710
FPCW$UNDERFLOW 1710
FPCW$UP 1710
FPCW$ZERODIVIDE 1710
FPU control word 1710
functions returning most recent run-time 2769, 2770
in multithread applications 1492
list of 361
locating run-time 359
loss of precision 1732
loss of precision errors 1732
methods of handling 357
option issuing for nonstandard Fortran 1063
option specifying maximum number of 547, 871
overflow errors 1732
rounding 1733
Run-Time Library 353
subroutine returning message for last detected 2712
underflow errors 1732
when building 331
ERRSNS 2642
ERR specifier for CLOSE 2457
ES 2056

escape sequence
C-style 1781
ETIME 2643
example programs
and traceback iinformation 415
exception handler
overriding 411
exception handling
option generating table of 581
exception parameters 1711
exceptions
debugging 164
locating source of 359
exclude code 1532
code coverage tool 1532
exclusive hint
option causing prefetches for stores with 787, 956
exclusive OR 1825, 2870
function performing 2870
executable
creating 110
saving compiler information in 104
executable statements 1746
execution
stopping program 3575
subroutine delaying for a program 3524
subroutine suspending for a process 3523
execution control 1883
execution environment routines 1398
execution flow 1453
execution mode 1402
EXIST 2109
EXIT 2645, 2646
exit behavior
function returning QuickWin 2753
function setting QuickWin 3445
exit codes
Fortran 358
exit parameters
function setting QuickWin 3445
ExitThread 1481
EXP 2647
explicit format 2031
explicit interface 295, 1935, 1937, 2923
Fortran array descriptor format 295
specifying 2923
when required 1937
explicit-shape arrays 1398, 1520, 1530, 1532, 1552, 1561, 1574, 1576, 1579, 1611, 1623, 1630, 1658, 1853
.dpi 1520, 1532, 1552, 1561
.dyn 1520, 1530, 1532, 1552, 1561, 1574, 1576, 1579
.spi 1532, 1552
formatted 1630
OpenMP* header 1398
optimizing 1611
pgopti.dpi 1532
pgopti.spi 1532
source 1530, 1658
unformatted 1630
explicit typing 1799
EXPONENT 2649
exponential procedures
table of 2269
exponential values
function returning 2647
exponents
function returning range of decimal 3363
expressions 696, 917, 1817, 1818, 1820, 1821, 1823, 1825, 1828, 1831, 2086, 2682, 3666
character 1823
data type of numeric 1821
effect of parentheses in numeric 1820
element array 2682
generalized masked array 2682
initialization 1828
logical 1825
masked array 3666
numeric 1818
option evaluating floating-point 696, 917
relational 1823
specification 1831
variable format 2086
extended intrinsic operators 1940
extensions
using 430
extent
function returning 3521
EXTERN 2384
EXTERNAL 1930, 2195, 2417, 2650
effect of block data program unit in 2417
effect on intrinsic procedures 1930
FORTRAN-66 interpretation of 2195
external data
mixed-language programming 285

external field separators 2066
external files 197, 1979, 3115
    associating with units 3115
    definition of 197
    preconnected units 197
external functions
    statement specifying entry point for 2627
external linkage with C 2414
external names
    option specifying interpretation of 744
external naming conventions
    mixed-language programming 274
external procedures 1745, 1897, 1918, 2148, 2334, 2650
    directive specifying alternate name for 2334
    interfaces of 1918
    using as actual arguments 2650
    using to open a file 2148
external subprograms 1745
external unit buffer
    subroutine flushing 2666
external unit number 6
    function writing a character to 3215
external user-defined names
    option appending underscore to 486
external user-written functions
    using to open files 2148

F

F 2050
    edit descriptor 2050
F_UFMTENDIAN environment variable 132
F90_dyncom routine 116
F90 files 121
FDATE 2652
FDX 2130
    value for CONVERT specifier 2130
FGETC 2653, 3250
    POSIX version of 3250
FGX 2130
    value for CONVERT specifier 2130
fields in common blocks 285
fields in record structures 1721
field width 2040, 2044, 2045, 2046, 2048, 2050, 2052
    for B descriptor 2045
    for D descriptor 2052
    for E descriptor 2052
field width *(continued)*
    for F descriptor 2050
    for I descriptor 2044
    for O descriptor 2046
    for Z descriptor 2048
FILE 2134
    specifier for OPEN 2134
file access methods 208
file access mode
    function setting 3448
file extensions
    option specifying additional Fortran 566
    option specifying for FPP 567
    option specifying for passage to linker 568
    output files 122
    supported by ifort command 121
file management procedures
    table of 2288
file name
    default 2134
filenames
    specifying default 227
file numeric format
    specifying 2130
file operation statements
    BACKSPACE 2407
    DELETE 2526
    ENDFILE 2605
    INQUIRE 2903
    OPEN 3115
    REWIND 3397
file operation statements in CLOSE 2457
file path
    function splitting into components 3528
file position
    functions returning 2702, 2785
    specifying in OPEN 2139
file position statements
    BACKSPACE 2407
    ENDFILE 2605
    REWIND 3397
file record length 222
file records 213
file record types 213
files
    accessing with INCLUDE 2895
    carriage control for terminal display 2129
    combining at compilation 2895
    disconnecting 2457

files *(continued)*
example of specifying name and pathname 358
function changing access mode of 2449
function deleting 2528
function finding specified 2656
function renaming 3387, 3388
function repositioning 2696
function returning full path of 2703
function returning information about 2697, 2754, 2986, 3564
function setting modification time for 3450
functions returning current position of 2702, 2785
function using path to delete 3637
input 121
internal 209
key 437
opening 3115
option specifying Fortran 1061
organization 208
repositioning to first record 3397
routine testing access mode of 2324
scratch 209
statement requesting properties of 2903
temporary 124
types of 1979
types of Microsoft* Fortran PowerStation compatible 246
using external user-written function to open 2148
file sharing 236, 2144
specifying 2144
file structure
specifying 2136
fill mask
functions using 2658, 2661
subroutine setting to new pattern 3451
fill shapes
subroutine returning pattern used to 2759
FIND 2655
FINDFILEQQ 2656
FIRSTPRIVATE 1342, 1346, 1371, 2521, 2579, 2657, 3139, 3145, 3146, 3418, 3520
in DEFAULT clause 2521
in DO directive 2579
in PARALLEL directive 3139
in PARALLEL DO directive 3145
in PARALLEL SECTIONS directive 3146
in SECTIONS directive 3418
in SINGLE directive 3520
in worksharing constructs 1371
FIRSTPRIVATE *(continued)*
summary of data scope attribute clauses 1342
using 1346
fixed-format source lines 1759
FIXEDFORMLINESIZE 140, 2657
equivalent compiler option for 140
fixed-length record type 213
fixed source format 588, 1757, 1759, 2191, 2657, 2695, 3098
directive for specifying 2695, 3098
directive setting line length for 2657
lines in 1759
option specifying file is 588
FLOAT 3371
FLOATI 3371
floating-point accuracy
option disabling optimizations affecting 698, 919
floating-point array operation 1694
floating-point calculations
option controlling semantics of 601, 606
Floating-point Compiler Options 1677
floating-point control procedures
table of 2273
floating-point control word 1706, 1710, 2732, 2959, 3415, 3438
subroutines returning 2732, 3415
subroutines setting 2959, 3438
floating-point conversion limitations 1721
floating-point data types 171, 181, 188, 1721, 1728, 1729, 1769
conversion limitations 1721
CRAY* big endian formats 181
digits of precision for REAL*4 1728
digits of precision for REAL*8 1729
IBM big endian formats 181
IEEE* big endian formats 181
IEEE* S_float 181, 1728
IEEE* T_float 181, 1728
methods of specifying nonnative formats 188
nonnative formats 181
normal and denormalized values of native formats 171
obtaining unformatted numeric formats 188
values for constants 171
VAX* D_float format 181
VAX* F_float format 181
VAX* G_float format 181
floating-point exception flags
function returning settings of 2672

floating-point exception flags *(continued)*
function setting 2674
floating-point exception handling for program
option allowing some control over 617
floating-point exception handling for routines
option allowing some control over 620
floating-point exceptions 1694, 1699, 1700, 1711, 1714
/fpe compiler option 1714
denormal exceptions 1694
exception parameters 1711
Fortran 1700
Fortran console applications 1714
Fortran DLL applications 1714
Fortran QuickWin applications 1714
Fortran Standard Graphics applications 1714
Fortran Windows applications 1714
types of 1714
floating-point inquiry procedures
table of 2273
floating-point numbers
exception parameters for 1711
formats for 1721
functions returning parameters 1725
loss of precision errors 1732
overflow errors 1732
overview 1675
overview of 1721
precision bits 1711
representation of 1724
rounding errors 1732, 1733
rounding flags 1711
special values 1722
underflow errors 1732
floating-point operations
option controlling semantics of 601, 606
option enabling combining of 593, 880
option improving consistency of 590
option rounding results of 611, 884
option specifying mode to speculate for 613, 886
Floating-point Operations
programming tradeoffs 1681
floating-point performance 1693
floating-point precision
option controlling for significand 821, 987
option improving for divides 825, 990
option improving for square root 826, 991
option improving general 737, 989
floating-point precision *(continued)*
option maintaining while disabling some optimizations 590
floating-point registers
option disabling use of high 637, 894
floating-point representation 1728
floating-point rounding flags 1711
floating-point stack 615, 888, 1721
option checking 615, 888
floating-point status word
parameters of 1709
setting and retrieving 1706
subroutine clearing exception flags in 2452
subroutines returning 2785, 3564
floating-point unit (FPU) 1706
FLOATJ 3371
FLOATK 3371
float-to-integer conversion
option enabling fast 1008, 1044
FLOODFILL 2658
FLOODFILL_W 2658
FLOODFILLRGB 2661
FLOODFILLRGB_W 2661
FLOOR 2663
flow dependency in loops 1584
fltconsistency compiler option 431
FLUSH 2664, 2666
flush-to-zero mode 1611
FMT 1984, 3365, 3673
specifier for READ 3365
specifier for WRITE 3673
focus
determining which window has 2902
setting 2667
FOCUSQQ 2667
font
function setting for OUTGTEXT 3455
function setting orientation angle for OUTGTEXT 3460
font characteristics
function returning 2762
font-related library functions 2762, 2764, 2899, 3130, 3455
fonts
function initializing 2899
function returning characteristics of 2762
function returning orientation of text for 2766
function returning size of text for 2764
function setting for OUTGTEXT 3455

fonts *(continued)*
function setting orientation angle for text 3460
FOR_ACCEPT environment variable 132
FOR_DEFAULT_PRINT_DEVICE environment variable 132
FOR_DESCRIPTOR_ASSIGN 2668
FOR_DIAGNOSTIC_LOG_FILE environment variable 132
FOR_DISABLE_DIAGNOSTIC_DISPLAY environment variable 132
FOR_DISABLE_STACK_TRACE environment variable 132
FOR_ENABLE_VERBOSE_STACK_TRACE environment variable 132
FOR_FMT_TERMINATOR environment variable 132
FOR_FULL_SRC_FILE_SPEC environment variable 132
FOR_GENERATE_DEBUG_EXCEPTION environment variable 132
FOR_GET_FPE 2672
FOR_IGNORE_EXCEPTIONS environment variable 132
for_iosdef.for file 358
FOR_NOERROR_DIALOGS environment variable 132
FOR_PRINT environment variable 132
for_rtl_finish_ 2673
for_rtl_init_ 2674
FOR_SET_FPE 2674
FOR_SET_REENTRANCY 2681
FOR_TYPE environment variable 132
FORALL 2682
FORCEINLINE 2384, 2385, 2388
option for ATTRIBUTES directive 2384, 2385, 2388
fordef.for file 1703
FOR files 121
FORM 2110, 2136
specifier for INQUIRE 2110
specifier for OPEN 2136
FORMAT 2031, 2685
specifications 2031
format control
terminating 2079
format lists 2031
format of data
default for list-directed output 2019
explicit 2031
implicit 2031
list-directed input 2000
list-directed output 2019
namelist input 2003
namelist output 2021
rules for numeric 2042
format of data *(continued)*
specifying file numeric 2130
using character string edit descriptors 2083
using control edit descriptors 2068
using data edit descriptors 2039
format of thread routines 1484
format specifications 2031, 2090
character 2031
interaction with i/o lists 2090
summary of edit descriptors 2031
format specifier 1984
FORMATTED 2110
specifier for INQUIRE 2110
formatted direct-access READ statements 2013
formatted direct-access WRITE statements 2026
formatted direct files 208
formatted files 208, 1630, 2136
direct-access 208
formatted i/o 2000, 2003, 2019, 2021, 2031
list-directed input 2000
list-directed output 2019
namelist input 2003
namelist output 2021
formatted records 1979, 2089
printing of 2089
formatted sequential files 208
formatted sequential READ statements 1999
formatted sequential WRITE statements 2018
forms for control edit descriptors 2068
forms for data edit descriptors 2040
forms for DO constructs 1886
FORT_BUFFERED environment variable 132, 1630
FORT_CONVERT_ext environment variable 132, 189
FORT_CONVERT.ext environment variable 132, 189
FORT_CONVERT environment variable 189, 190
FORT_CONVERTn environment variable 132, 190
fortcom.exe file 437
FORTn environment variable 132
Fortran 2003 features 1741
FORTRAN 66
option applying semantics of 570
FORTRAN-66 interpretation of EXTERNAL 2195
FORTRAN 77
option using run-time behavior of 572
option using semantics for kind parameters 690
FORTRAN 77 language standard 429
Fortran 90
directive enabling or disabling extensions to 3103, 3577

Fortran 90 *(continued)*
obsolescent features in 2193
Fortran 90 language standard 187, 429
using RECL units for unformatted files 187
Fortran 95
deleted features in 2191
directive enabling or disabling extensions to 3103, 3577
obsolescent features in 2191
Fortran 95/90 character set 1750
Fortran 95/90 pointers 3163
Fortran 95 language standard 429
Fortran and Assembly
summary of programming issues 264
Fortran and C/C++*
data types 289
external names 275
summary of programming issues 264
using interface blocks 279
Fortran and MASM
data types 289
external names 275
using interface blocks 279
Fortran array descriptor format 295
Fortran characters 1745
Fortran console applications
floating-point exceptions 1714
Fortran DLL applications
floating-point exceptions 1714
Fortran executables
creating 257
Fortran Language Standards 429
Fortran PowerStation
compatibility with 246
Fortran preprocessor (FPP)
list of options 1232
option affecting end-of-line comments 476
option defining symbol for 153, 522
option passing options to 1111
option running on files 625, 889
option sending output to a file 827
option sending output to stdout 561, 562
Fortran procedures
tables of 2248
Fortran programs
debugging 161
Fortran QuickWin applications
floating-point exceptions 1714
Fortran source files
compiling 99, 108
Fortran Standard Graphics applications
floating-point exceptions 1714
Fortran standards
and extensions 430
Fortran Standard type aliasability rules
option affecting adherence to 479, 847
Fortran statements
tables of 2248
Fortran Windows* applications
floating-point exceptions 1714
FP_CLASS 2691
FPATH environment variable 129
fpe compiler option 359
fpp
directives 146
introduction 143
fpp.exe file 437
fpp files
option to keep 143
fpp options
list of 1232
FPSW$DENORMAL 1709
FPSW$INEXACT 1709
FPSW$INVALID 1709
FPSW$MSW_EM 1709
FPSW$OVERFLOW 1709
FPSW$UNDERFLOW 1709
FPSW$ZERODIVIDE 1709
FPU rounding control
option setting to truncate 1009, 1045
FPUTC 2692, 3256
POSIX version of 3256
FRACTION 2693
FREE 2694
FREEFORM 140, 2695, 3098
equivalent compiler option for 140
free source format 638, 1754, 2695, 3098
directive specifying 2695, 3098
option specifying file is 638
FSEEK 2696, 3257
POSIX version of 3257
FSTAT 2697, 3258
POSIX version of 3258
FTELL 2702, 3258
POSIX version of 3258
FTELLI8 2702
ftrapuv compiler option 359

FTZ flag 1689
FTZ mode 1611
FULLPATHQQ 2703
FUNCTION 2705
function entry and exit points
    option determining instrumentation of 586, 912
function expansion 1514
function grouping
    option enabling or disabling 833
function grouping optimization 1566
function ordering optimization 1566
function order list 1520, 1573
    enabling or disabling 1520
function order lists 1566
function preemption 1511
function profiling
    option compiling and linking for 805
function references 1916
function result 1853, 3392
    as explicit-shape array 1853
    specifying different name for 3392
functions 574, 882, 1897, 1915, 1916, 1944, 1947, 2429, 2650, 2705, 3377, 3392, 3569
    defining in a statement 3569
    effect of ENTRY in 1944
    elemental intrinsic 1947
    ELEMENTAL keyword in 2705
    EXTERNAL 2650
    general rules for 1915
    generic 1947
    inquiry 1947
    invoking 1916
    invoking in a CALL statement 2429
    not allowed as actual arguments 1947
    option aligning on byte boundary 574, 882
    PURE keyword in 2705
    RECURSIVE keyword in 2705, 3377
    references to 1916
    RESULT keyword in 2705, 3392
    specific 1947
    statement 3569
    that apply to arrays 1947
    transformational 1947
functions not allowed as actual arguments
    table of 1947
function splitting
    option enabling or disabling 597, 883

G

G 2058
    edit descriptor 2058
    effect of data magnitude on format conversions 2058
gcc C++ run-time libraries
    include file path 669
    option adding a directory to second 669
    option specifying to link to 520
GCCROOT environment variable 129
general compiler directives 1260, 1310, 1453, 1459, 1461, 1511, 1519, 1520, 1573, 1591, 1595, 1602, 1605, 1611, 1636, 2159, 2160, 2253
    affecting data prefetches 1591, 1602
    affecting software pipelining 1591, 1605
    for auto-parallelization 1453
    for IA-32 architecture 1611
    for improving run-time efficiency 1636
    for inlining functions 1511
    for profile-guided optimization 1519
    for vectorization 1459, 1461
    instrumented code 1520
    processor-specific code 1310
    profile-optimized executable 1520
    profiling information 1573
    reports 1260
    rules for 2159
    table of 2253
general directives 2160, 2334, 2362, 2371, 2518, 2522, 2544, 2657, 2695, 2833, 2884, 2916, 2949, 2984, 3026, 3098, 3099, 3100, 3103, 3105, 3108, 3109, 3114, 3123, 3124, 3136, 3142, 3143, 3189, 3207, 3370, 3577, 3592, 3632, 3643, 3644, 3654, 3655, 3657, 3658, 3659
    ALIAS 2334
    ASSUME_ALIGNED 2362
    ATTRIBUTES 2371
    DECLARE 2518
    DEFINE 2522, 3632
    DISTRIBUTE POINT 2544
    ENDIF 2884
    FIXEDFORMLINESIZE 2657
    FREEFORM 2695, 3098
    IDENT 2833
    IF 2884
    IF DEFINED 2884
    INTEGER 2916

general directives *(continued)*
IVDEP 2949
LOOP COUNT 2984
MESSAGE 3026
NODECLARE 2518
NOFREEFORM 2695, 3098
NOOPTIMIZE 3099, 3123
NOPARALLEL 3142, 3143
NOPREFETCH 3100, 3189
NOSTRICT 3103, 3577
NOSWP (i64) 3105, 3592
NOUNROLL 3108, 3643
NOUNROLL_AND_JAM 3644
NOVECTOR 3109, 3655
OBJCOMMENT 3114
OPTIMIZE 3099, 3123
OPTIONS 3124
PACK 3136
PARALLEL 3142, 3143
PREFETCH 3100, 3189
PSECT 3207
REAL 3370
STRICT 3103, 3577
SWP (i64) 3105, 3592
UNDEFINE 2522, 3632
UNROLL 3108, 3643
UNROLL_AND_JAM 3644
VECTOR ALIGNED 3654, 3659
VECTOR ALWAYS 3109, 3655
VECTOR NONTEMPORAL (i32, i64em) 3657, 3658
VECTOR TEMPORAL (i32, i64em) 3657, 3658
VECTOR UNALIGNED 3654, 3659
generalized editing (G) 2058
general rules for numeric editing 2042
generic assignment 1942
generic identifier 2923
generic interface 1938, 2923
generic intrinsic functions
references to 1930
generic name 2176, 2923
references to 2176
generic operators 1940
generic procedures 1930, 1938, 2175
references to 1930
references to intrinsic 1930
unambiguous references to 2175
generic references 2176
gen-interfaces compiler option 331
GERROR 2712
GET_COMMAND 2730
GET_COMMAND_ARGUMENT 2731
GET_ENVIRONMENT_VARIABLE 2745
GETACTIVEQQ 2714
GETARCINFO 2714
GETARG 2716, 3259
POSIX version of 3259
GETBKCOLOR 2718
GETBKCOLORRGB 2719
GETC 2722, 3260
POSIX version of 3260
GETCHARQQ 2723
GETCOLOR 2725
GETCOLORRGB 2727
GETCONTROLFPQQ 1710, 1711, 2732
example of 1711
using to return the control word value 1710
GETCURRENTPOSITION 2735
GETCURRENTPOSITION_W 2735
GetCurrentProcess 1495
GetCurrentProcessId 1495
GetCurrentThreadId 1481
GETCWD 2737, 3261
POSIX version of 3261
GETDAT 2738
GETDRIVEDIRQQ 2740
GETDRIVESIZEQQ 2741
GETDRIVESQQ 2744
GETENV 2745
GETENVQQ 2748
GETEXCEPTIONPTRSQQ 2751
GetExitCodeProcess 1495
GetExitCodeThread 1481
GETEXITQQ 2753
GETFILEINFOQQ 2754
GETFILLMASK 2759
GETFONTINFO 2762
GETGID 2764
GETGTEXTEXTENT 2764
GETGTEXTROTATION 2766
GETHWNDQQ 2767
GETIMAGE 2768, 2888
function returning memory needed for 2888
GETIMAGE_W 2768
GetLastError 1492
GETLASTERROR 2769
GETLASTERRORQQ 2770
GETLINESTYLE 2772
GETLOG 2774

GETPHYSCOORD 2775
GETPID 2777
GETPIXEL 2777
GETPIXEL_W 2777
GETPIXELRGB 2779
GETPIXELRGB_W 2779
GETPIXELS 2781
GETPIXELSRGB 2782
GETPOS 2785
GETPOSI8 2785
GETSTATUSFPQQ 2785
GETSTRQQ 2787
GETTEXTCOLOR 2789
GETTEXTCOLORRGB 2790
GETTEXTPOSITION 2792
GETTEXTWINDOW 2793
GetThreadPriority 1481
GETTIM 2795
GETTIMEOFDAY 2796
GETUID 2796
GETUNITQQ 2797
GETVIEWCOORD 2798
GETVIEWCOORD_W 2798
GETWINDOWCONFIG 2799
GETWINDOWCOORD 2804
GETWRITEMODE 2805
GETWSIZEQQ 2806
global entities 1748
global external Fortran data
    mixed-language programming 285
global scope 2171
glossary
    A 3679
    B 3682
    C 3683
    D 3685
    E 3689
    F 3690
    G 3692
    H 3692
    I 3693
    K 3695
    L 3695
    M 3697
    N 3698
    O 3699
    P 3700
    Q 3702
    R 3702
glossary *(continued)*
    S 3704
    T 3708
    U 3709
    V 3710
    W 3710
    Z 3711
GMTIME 2808
GOTO 2810, 2811, 2813
GO TO 2810, 2811, 2813
    assigned 2810
    computed 2811
    unconditional 2813
graphics applications
    option creating and linking 1108
graphics output
    function returning background color index for 2718
    function returning background RGB color for 2719
    function setting background color index for 3428
    function setting background RGB color for 3429
    subroutine limiting to part of screen 3431
graphics position
    subroutine moving to a specified point 3058
    subroutine returning coordinates for current 2735
graphics routines
    ARC and ARC_W 2348
    CLEARSCREEN 2451
    DISPLAYCURSOR 2543
    ELLIPSE and ELLIPSE_W 2597
    FLOODFILL and FLOODFILL_W 2658
    FLOODFILLRGB and FLOODFILLRGB_W 2661
    function returning status for 2814
    GETARCINFO 2714
    GETBKCOLOR 2718
    GETBKCOLORRGB 2719
    GETCOLOR 2725
    GETCOLORRGB 2727
    GETCURRENTPOSITION and
    GETCURRENTPOSITION_W 2735
    GETFILLMASK 2759
    GETFONTINFO 2762
    GETGTEXTEXTENT 2764
    GETGTEXTROTATION 2766
    GETIMAGE and GETIMAGE_W 2768
    GETLINESTYLE 2772
    GETPHYSCOORD 2775
    GETPIXEL and GETPIXEL_W 2777
    GETPIXELRGB and GETPIXELRGB_W 2779
    GETPIXELS 2781

graphics routines *(continued)*
GETPIXELSRGB 2782
GETTEXTCOLOR 2789
GETTEXTCOLORRGB 2790
GETTEXTPOSITION 2792
GETTEXTWINDOW 2793
GETVIEWCOORD and GETVIEWCOORD_W 2798
GETWINDOWCOORD 2804
GETWRITEMODE 2805
GRSTATUS 2814
IMAGESIZE and IMAGESIZE_W 2888
INITIALIZEFONTS 2899
LINETO and LINETO_W 2966
LINETOAR 2968
LINETOAREX 2970
LOADIMAGE and LOADIMAGE_W 2976
MOVETO and MOVETO_W 3058
OUTGTEXT 3130
OUTTEXT 3133
PIE and PIE_W 3160
POLYBEZIER and POLYBEZIER_W 3169
POLYBEZIERTO and POLYBEZIERTO_W 3175
POLYGON and POLYGON_W 3181
POLYLINEQQ 3185
PUTIMAGE and PUTIMAGE_W 3216
RECTANGLE and RECTANGLE_W 3374
REMAPALLPALETTERGB and REMAPPALETTERGB 3384
SAVEIMAGE and SAVEIMAGE_W 3408
SCROLLTEXTWINDOW 3412
SETBKCOLOR 3428
SETBKCOLORRGB 3429
SETCLIPRGN 3431
SETCOLOR 3434
SETCOLORRGB 3436
SETFILLMASK 3451
SETFONT 3455
SETGTEXTROTATION 3460
SETLINESTYLE 3462
SETPIXEL and SETPIXEL_W 3469
SETPIXELRGB and SETPIXELRGB_W 3470
SETPIXELS 3473
SETPIXELSRGB 3474
SETTEXTCOLOR 3477
SETTEXTCOLORRGB 3478
SETTEXTCURSOR 3480
SETTEXTPOSITION 3483
SETTEXTWINDOW 3484
SETVIEWORG 3487
graphics routines *(continued)*
SETVIEWPORT 3488
SETWINDOW 3489
SETWRITEMODE 3498
table of 2282
WRAPON 3671
graphics viewport
subroutine redefining 3488
Greenwich mean time
function returning seconds and milliseconds since 2796
function returning seconds since 3403
subroutine returning 2808
group ID
function returning 2764
GRSTATUS 2814
GXX_INCLUDE environment variable 129
GXX_ROOT environment variable 129

H

H 2084
edit descriptor 2084
HABS 2321
handle
function returning unit number of window 2797
handlers
function establishing for IEEE exceptions 2867
HBCLR 2825
HBITS 2826
HBSET 2827
HDIM 2539
help
using in Microsoft Visual Studio* 79
heuristics
affecting software pipelining 2384, 2385, 2388
for inlining functions 2384, 2385, 2388
overriding optimizer efficiency 3654, 3659
overriding vectorizer efficiency 3109, 3655
hexadecimal constants 1792, 1793, 2199
alternative syntax for 2199
hexadecimal editing (Z) 2048
hexadecimal values
transferring 2048
HFIX 2910
HIAND 2821
hidden-length character arguments
option specifying convention for passing 670

HIEOR 2870
high-level optimization 1273
high-level optimizer 1260, 1581
high performance 1239
high performance programming 1242, 1260, 1310, 1519, 1581, 1611, 1630
  applications for 1581
  dispatch options for 1310
  guidelines for 1611
  improving performance 1630
  list 1630
  options for 1310
  parsing 1630
  performance 1630
  processors for 1242, 1310
  report generation 1260
HIOR 2929
HIXOR 2870
HLO 1273, 1581
  reports 1273
HMOD 3040
HMVBITS 3062
HNOT 3106
Hollerith arguments 1927
Hollerith constants 179, 1792, 1794
  representation of 179
Hollerith editing 2084
Hollerith values
  transferring 2062
host association 2183
host computer name
  function returning 2819
HOSTNAM 2819
HOSTNM 2819
hotness threshold
  option setting 838, 1000
hot patching
  option preparing a routine for 666
hotspots 1247
HSHFT 2943
HSHFTC 2945
HSIGN 3509
HTEST 2422
HUGE 2820
hyperbolic arccosine
  function returning 2328
hyperbolic arcsine
  function returning 2352
hyperbolic arctangent
  function returning 2368
hyperbolic cosine
  function returning 2488
hyperbolic sine
  function returning 3513
hyperbolic tangent
  function returning 3599
Hyper-Threading Technology
  parallel loops 1454
  thread pools 1454

I

I 2044
  edit descriptor 2044
I/O
  asynchronous 253
  choosing optimal record type 213
  data formats for unformatted files 181
  file sharing 236
  list-directed input 2000
  list-directed output 2019
  namelist input 2003
  namelist output 2021
  record 213
  specifying record length for efficiency 213
I/O buffers
  flushing and closing 2319
I/O control list 1981, 1983, 1984, 1985, 1986, 1988, 1989
  advance specifier 1988
  asynchronous specifier 1988
  branch specifiers 1986
  character count specifier 1989
  format specifier 1984
  I/O status specifier 1985
  id specifier 1989
  namelist specifier 1985
  pos specifier 1989
  record specifier 1985
  unit specifier 1983
I/O editing
  overview of 2031
I/O error conditions
  subroutine returning information on 2642
I/O formatting 2031

I/O lists 1990, 1991, 1995, 2090
how to specify 1990
implied-do lists in 1995
interaction with format specifications 2090
simple list items in 1991
I/O procedures
table of 2252
I/O statements
ACCEPT 2323
BACKSPACE 2407
DELETE 2526
ENDFILE 2605
forms of 203
INQUIRE 2903
list of 201
OPEN 3115
PRINT 3194
READ 3365
REWIND 3397
REWRITE 3398
WRITE 3673
I/O statements in CLOSE 2457
I/O statement specifiers 235
I/O status specifier 1985
I/O units 1983
IA-32 architecture based applications
HLO 1581
methods of parallelization 1242
options 1306, 1310
targeting 1306, 1310
using intrinsics in 1638
IA-64 architecture based applications
auto-vectorization in 1242
HLO 1581
methods of parallelization 1242
options 1313
pipelining for 1605
report generation 1260
targeting 1313
using intrinsics in 1638
IABS 2321
IACHAR 2820
IADDR 2409
IAND 2821
IARG 2823
IARGC 2823
ias.exe file 437
IBCHNG 2824
IBCLR 2825
IBITS 2826
IBM 2130
value for CONVERT specifier 2130
IBM* character set 2213
IBSET 2827
ICHAR 2829
ID 1989, 2111, 3365, 3673
specifier for INQUIRE 2111
specifier for READ 3365
specifier for WRITE 3673
IDATE 2830, 2831
IDATE4 2832
IDB 164
IDB (see Intel(R) Debugger) 161
IDENT 2833
IDFLOAT 2833
IDIM 2539
IDINT 2910
idis.exe file 437
IDNINT 3071
IEEE_ARITHMETIC 1908
IEEE_CLASS 2834
IEEE_COPY_SIGN 2835
IEEE_EXCEPTIONS 1909
IEEE_FEATURES 1910
IEEE_FLAGS 2861
IEEE_GET_FLAG 2835
IEEE_GET_HALTING_MODE 2836
IEEE_GET_ROUNDING_MODE 2837
IEEE_GET_STATUS 2838
IEEE_GET_UNDERFLOW_MODE 2839
IEEE_HANDLER 2867
IEEE_IS_FINITE 2839
IEEE_IS_NAN 2840
IEEE_IS_NEGATIVE 2841
IEEE_IS_NORMAL 2841
IEEE_LOGB 2842
IEEE_NEXT_AFTER 2843
IEEE_REM 2844
IEEE_RINT 2844
IEEE_SCALB 2845
IEEE_SELECTED_REAL_KIND 2846
IEEE_SET_FLAG 2847
IEEE_SET_HALTING_MODE 2847
IEEE_SET_ROUNDING_MODE 2848
IEEE_SET_STATUS 2849
IEEE_SET_UNDERFLOW_MODE 2850
IEEE_SUPPORT_DATATYPE 2851
IEEE_SUPPORT_DENORMAL 2852

IEEE_SUPPORT_DIVIDE 2852
IEEE_SUPPORT_FLAG 2853
IEEE_SUPPORT_HALTING 2854
IEEE_SUPPORT_INF 2854
IEEE_SUPPORT_IO 2855
IEEE_SUPPORT_NAN 2856
IEEE_SUPPORT_ROUNDING 2856
IEEE_SUPPORT_SQRT 2857
IEEE_SUPPORT_STANDARD 2858
IEEE_SUPPORT_UNDERFLOW_CONTROL 2859
IEEE_UNORDERED 2860
IEEE_VALUE 2860
IEEE*
  floating-point formats 1724
  floating-point values 1722
  nonnative big endian data 181
  S_float data 1728
  S_float data ranges 171
  S_float representation (COMPLEX*8) 1730
  T_float data 1728, 1729
  T_float data ranges 171
  T_float representation (COMPLEX*16) 1731
  T_float representation (COMPLEX*32) 1731
  X_floating format 1729
IEEE* exceptions
  function clearing status of 2861
  function establishing a handler for 2867
  function getting or setting status of 2861
IEEE* flags
  function clearing 2861
  function getting or setting 2861
IEEE* floating-point representation 1728
IEEE* intrinsic modules
  function an integer value rounded according to the current rounding mode 2844
  function assigning a value to an exception flag. 2847
  function creating IEEE value 2860
  function restoring state of the floating-point environment 2849
  function returning argument with copied sign 2835
  function returning exponent of radix-independent floating-point number 2845
  function returning FP value equal to unbiased exponent of argument 2842
  function returning IEEE class 2834
  function returningnext representable value after X toward Y 2843
IEEE* intrinsic modules *(continued)*
  function returning result value from a remainder operation 2844
  function returning value of the kind parameter of an IEEE REAL data type 2846
  function returning whether expection flag is signaling 2835
  function returning whether IEEE value is finite 2839
  function returning whether IEEE value is negative 2841
  function returning whether IEEE value is normal 2841
  function returning whether IEEE value is Not-a-number(NaN) 2840
  function returning whether one or more of the arguments is Not-a-Number (NaN) 2860
  function returning whether processor supports ability to control the underflow mode 2859
  function returning whether processor supports IEEE arithmetic 2851
  function returning whether processor supports IEEE base conversion rounding during formatted I/O 2855
  function returning whether processor supports IEEE denormalized numbers 2852
  function returning whether processor supports IEEE divide 2852
  function returning whether processor supports IEEE exceptions 2853
  function returning whether processor supports IEEE features defined in the standard 2858
  function returning whether processor supports IEEE halting 2854
  function returning whether processor supports IEEE infinities 2854
  function returning whether processor supports IEEE Not-a-Number feature 2856
  function returning whether processor supports IEEE rounding mode 2856
  function returning whether processor supports IEEE SQRT 2857
  function setting current underflow mode 2850
  function storing current rounding mode 2837
  function storing current state of floating-point environment 2838
  function storing current underflow mode 2839
  function storing halting mode for exception 2836
  function that controls halting or continuation after an exception. 2847

IEEE* intrinsic modules *(continued)*
function that sets rounding mode. 2848
IEEE* numbers
function testing for NaN values 2948
IEEE equivalent function
IEEE logb function 2842
IEEE nextafter function 2843
IEEE rem function 2844
IEEE scalb function 2845
IEEE unordered 2860
IEEE intrinsic modules
IEEE_ARITHMETIC 1908
IEEE_EXCEPTIONS 1909
IEEE_FEATURES 1910
Quick Reference Tables 1911
IEEE intrinsic modules and procedures 1906
IEOR 2870
IERRNO 2712, 2871
subroutine returning message for last error detected by 2712
If 2873
IF 2873, 2875, 2876, 2884, 3139, 3145, 3146
arithmetic 2873
clause in PARALLEL directive 3139
clause in PARALLEL DO directive 3145
clause in PARALLEL SECTIONS directive 3146
directive for conditional compilation 2884
logical 2875
IF DEFINED 2884
IFIX 2910
ifl.exe file 437
IFLOATI 2887
IFLOATJ 2887
ifmt.mod 1481
ifort
output files 122
ifort.cfg file 437
ifort.exe file 437
ifortcfg environment variable 129
ifort command
examples 109
redirecting output 113
requesting listing file using 103
syntax for 108
using 107
ifortvars.bat file 331, 437
ifortvars.sh file 331
IFPORT portability module
overview 325
IFPORT portability module *(continued)*
using 325
IGNORE_LOC 2385
option for ATTRIBUTES directive 2385
IIABS 2321
IIAND 2821
IIBCLR 2825
IIBITS 2826
IIBSET 2827
IIDIM 2539
IIDINT 2910
IIDNNT 3071
IIEOR 2870
IIFIX 2910
IINT 2910
IIOR 2929
IIQINT 2910
IIQNNT 3071
IISHFT 2943
IISHFTC 2945
IISIGN 3509
IIXOR 2870
IJINT 2910
ILEN 2888
ILO 1260
IMAG 2330
images
function displaying from bitmap file 2976
function returning storage size of 2888
function saving into Windows bitmap file 3408
transferring from memory to screen 3216
IMAGESIZE 2888
IMAGESIZE_W 2888
IMAX0 2995
IMAX1 2995
IMIN0 3028
IMIN1 3028
IMOD 3040
IMPLICIT 1930, 2889
effect on intrinsic procedures 1930
implicit format 2031
implicit interface 1935, 2923
IMPLICIT NONE 2889
implicit typing 1800, 2889
overriding default 2889
implied-DO lists 1995
implied-DO loop 1636, 1995
list in i/o lists 1995
IMPORT 2891

improving 1345, 1630, 1636
code 1636
I/O performance 1630
run-time performance 1636
IMSL* libraries
option letting you link to 895
IMVBITS 3062
INCHARQQ 2892
INCLUDE 102, 260, 2895
directory searched for filenames 102, 260
INCLUDE environment variable 129
include file path
option adding a directory to 667
option removing standard directories from 1116
INCLUDE files
option specifying directory to search for 129
searching for 260
using 260
INCLUDE lines 2895
including files during compilation 2895
inclusive OR 1825, 2929
function performing 2929
incremental linking
linker option specifying treatment of 413
INDEX 2898
index for last occurrence of substring
function locating 3401
ININT 3071
initialization expressions 1785, 1828, 3626
for derived-type components 1785
inquiry functions allowed in 1828
in type declaration statements 3626
transformational functions allowed in 1828
initialization values for reduction variables 1348
INITIALIZEFONTS 2899
initializing variables 2500
INITIALSETTINGS 2900
INLINE 2384, 2385, 2388
option for ATTRIBUTES directive 2384, 2385, 2388
inlined code
option producing source position information for 676, 896
inline function expansion
option disabling 749
option specifying level of 674, 680, 758
inlining 585, 679, 682, 683, 685, 687, 688, 694, 695, 900, 901, 903, 905, 906, 908, 915, 916, 1511, 1512, 1514, 1519, 1636, 1658
compiler directed 1512
inlining *(continued)*
developer directed 1514
option disabling full and partial 694, 915
option disabling partial 695, 916
option forcing 679, 900
option specifying lower limit for large routines 685, 905
option specifying maximum size of function for 585
option specifying maximum times for a routine 683, 903
option specifying maximum times for compilation unit 682, 901
option specifying total size routine can grow 687, 906
option specifying upper limit for small routine 688, 908
preemption 1511
inlining options
option specifying percentage multiplier for 677, 898
INMAX 2901
INOT 3106
input/output editing 2031
input/output lists 1990
input/output statements 2095
input and output files 121
input and output procedures
table of 2252
input data
terminating short fields of 2066
input file extensions 121
input statements for data transfer
ACCEPT 2323
READ 3365
INQFOCUSQQ 2902
INQUIRE 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2903
ACCESS specifier 2104
ACTION specifier 2104
ASYNCHRONOUS specifier 2105
BINARY specifier 2105
BLANK specifier 2106
BLOCKSIZE specifier 2106
BUFFERED specifier 2106
CARRIAGECONTROL specifier 2107
CONVERT specifier 2107
DELIM specifier 2108
DIRECT specifier 2109

INQUIRE *(continued)*
EXIST specifier 2109
FORMATTED specifier 2110
FORM specifier 2110
ID specifier 2111
IOFOCUS specifier 2111
MODE specifier 2112
NAMED specifier 2112
NAME specifier 2112
NEXTREC specifier 2113
NUMBER specifier 2113
OPENED specifier 2113
ORGANIZATION specifier 2114
PAD specifier 2114
PENDING specifier 2114
POSITION specifier 2116
POS specifier 2115
READ specifier 2116
READWRITE specifier 2117
RECL specifier 2117
RECORDTYPE specifier 2117
SEQUENTIAL specifier 2118
SHARE specifier 2119
UNFORMATTED specifier 2120
WRITE specifier 2120
INQUIRE statement 232
inquiry functions 1947, 2340, 2360, 2416, 2429, 2468, 2538, 2629, 2635, 2820, 2823, 2888, 2914, 2955, 2958, 2961, 2977, 2997, 3030, 3066, 3070, 3189, 3192, 3350, 3363, 3504, 3521, 3522, 3612
ALLOCATED 2340
ASSOCIATED 2360
BIT_SIZE 2416
CACHESIZE 2429
COMMAND_ARGUMENT_COUNT 2468
DIGITS 2538
EOF 2629
EPSILON 2635
for argument presence 3192
for arrays 2340, 2958, 3504, 3521
for bits 2416
for character length 2961
for numeric models
DIGITS 2538
EPSILON 2635
HUGE 2820
MAXEXPONENT 2997
MINEXPONENT 3030
inquiry functions *(continued)*
for numeric models *(continued)*
PRECISION 3189
RADIX 3350
RANGE 3363
TINY 3612
for pointers 2360
HUGE 2820
IARGC 2823
ILEN 2888
INT_PTR_KIND 2914
KIND 2955
LBOUND 2958
LEN 2961
LOC 2977
MAXEXPONENT 2997
MINEXPONENT 3030
NARGS 3066
NEW_LINE 3070
PRECISION 3189
PRESENT 3192
RADIX 3350
RANGE 3363
SHAPE 3504
SIZE 3521
SIZEOF 3522
TINY 3612
INSERTMENUQQ 2907
instruction-level parallelism 1242
instrumentation 1021, 1075, 1519, 1520, 1530, 1576, 1660
compilation 1530
execution 1530
feedback compilation 1530
generating 1520
option enabling or disabling for specified functions 1021, 1075
preventing aliasing 1660
program 1519
INT 2910
INT_PTR_KIND 2914
INT1 2910
INT2 2910
INT4 2910
INT8 2910
INTC 2913
INTEGER 140, 1765, 2915, 2916
compiler directive 2916
equivalent compiler option for 140

INTEGER *(continued)*
type 1765, 2915
INTEGER(1) 1765
INTEGER(2) 1765
INTEGER(4) 1765
INTEGER(8) 1765
INTEGER(KIND=1) representation 175
INTEGER(KIND=2) representation 175
INTEGER(KIND=4) representation 176
INTEGER(KIND=8) representation 176
INTEGER*1 1765
INTEGER*2 1765
INTEGER*4 1765
INTEGER*8 1765
integer constants 1765, 1766
integer data
function returning kind type parameter for 3422
model for 2224
integer data representations 174
integer data type 171, 174, 181, 188, 1765, 1766, 2186, 2910, 2915
constants 1766
declarations and options 171, 174
default kind 1765
function converting to 2910
methods of specifying endian format 188
nonnative formats 181
ranges 1765
storage 2186
integer edit descriptors 2044
integer editing (I) 2044
INTEGER KIND to hold address
function returning 2914
integer model 2224, 2820, 3612
function returning largest number in 2820
function returning smallest number in 3612
integer pointers 294, 1010, 1051, 3166
mixed-language programming 294
option affecting aliasing of 1010, 1051
integers
converting to RGB values 3399
directive specifying default kind 2916
function converting KIND=2 to KIND=4 2983
function converting KIND=4 to KIND=2 3508
function converting to quad-precision type 3346
function converting to single-precision type 2887, 3371
function multiplying two 64-bit signed 3061
function multiplying two 64-bit unsigned 3060
integers *(continued)*
function performing bit-level test for 2416
function returning difference between 2539
function returning leading zero bits in 2960
function returning maximum positive 2901
function returning number of 1 bits in 3187
function returning parity of 3188
function returning trailing zero bits in 3615
function returning two's complement length of 2888
functions converting to double-precision type 2536, 2537, 2833
models for data 2224
subroutine performing bit-level set and clear for 2412
INTEGERTORGB 2917
INTEL_LICENSE_FILE environment variable 129
INTEL_PROF_DUMP_CUMULATIVE environment variable 1574
INTEL_PROF_DUMP_INTERVAL environment variable 1574
Intel(R) 64 applications 115
Intel(R) 64 architecture based applications
HLO 1581
methods of parallelization 1242
options 1306, 1310
targeting 1306, 1310
using intrinsics in 1638
Intel(R) architectures 1611
Intel(R) compatibility libraries for OpenMP* 1321
Intel(R) compiler-generated code 1658
Intel(R) Debugger 161, 168
Intel(R)-extended intrinsics 1638
Intel(R) extension environment variables 1383
Intel(R) extension routines 1402
Intel(R) Fortran
compiler options 137
file extensions passed to compiler 121
handling run-time errors 351
intrinsic data types 1763
portability considerations 429
running Fortran applications 118
using the debugger 161
Intel(R) Fortran character set 1750
Intel(R) Fortran Compiler command prompt window 95
Intel(R) Fortran language extensions 2233
Intel(R) linking tools 1497
Intel(R) MKL
option letting you link to 732, 933

Intel(R) Trace Collector API
option inserting probes to call 1020, 1073
Intel(R) Visual Fortran
creating multithread applications 1479
Intel-provided libraries
option linking dynamically 1058
option linking statically 1068
INTENT 2919
intent of arguments 2919
interaction between format specifications and i/o lists 2090
INTERFACE 2923
INTERFACE ASSIGNMENT 2923
interface blocks 279, 657, 1839, 1938, 1940, 1942, 2923, 3047, 3053
for generic names 1938
for mixed-language programming 279
generic identifier in 2923
module procedures in 3047, 3053
option generating for routines 657
pure procedures in 2923
using ASSIGNMENT(=) 1839
using generic assignment in 1942
using generic operators in 1940
using generic procedures in 1938
INTERFACE OPERATOR 2923
interfaces 295, 1918, 1929, 1935, 1937, 1938, 2923
and Fortran array descriptor format 295
explicit 1935, 2923
generic 1938
implicit 1935, 2923
of dummy procedures 1929
of external procedures 1918
of internal procedures 1918
procedures requiring explicit 1937
INTERFACE TO 2926
intermediate files
option saving during compilation 1013, 1054
intermediate language scalar optimizer 1260
intermediate representation (IR) 1497, 1501
intermediate results 1630
using memory for 1630
internal address
function returning 2977
internal files 197, 209, 1918, 1979
interfaces of 1918
rules for using 197
internal procedures 261, 1745, 1897, 1918, 2483
advantages of 261
internal READ statements 2014
rules for 2014
internal subprograms 1636, 1745, 2483
CONTAINS statement 2483
internal WRITE statements 2027
rules for 2027
interoperability with C 306, 2414
interprocedural optimizations 100, 584, 693, 699, 914, 920, 1260, 1263, 1497, 1500, 1501, 1504, 1506, 1508, 1512, 1519, 1578, 1658
capturing intermediate output 1501
code layout 1506
compilation 1497
compiling 1501
compiling and linking for 100
considerations 1504
creating libraries 1508
initiating 1578
issues 1504
large programs 1504
linking 1497, 1501
option enabling additional 693, 914
option enabling between files 699, 920
option enabling for single file compilation 584
options 1500
overview 1497
performance 1504
reports 1263
using 1501
whole program analysis 1497
xiar 1508
xild 1508
xilibtool 1508
interrupt signal
registering a function to call for 3516
interrupt signal handling
function controlling 3513
INTRINSIC 2927
intrinsic assignment 1833, 1834, 1836, 1837, 1838, 2357
array 1838
character 1836
derived-type 1837
logical 1836
numeric 1834
intrinsic data types 1763, 2019, 2186
default formats for list-directed output 2019
storage requirements for 2186

intrinsic function
IEEE_CLASS 2834
IEEE_IS_FINITE 2839
IEEE_IS_NAN 2840
IEEE_IS_NEGATIVE 2841
IEEE_IS_NORMAL 2841
IEEE_LOGB 2842
IEEE_NEXT_AFTER 2843
IEEE_REM 2844
IEEE_RINT 2844
IEEE_SCALB 2845
IEEE_SELECTED_REAL_KIND 2846
IEEE_SET_FLAG 2847
IEEE_SET_HALTING_MODE 2847
IEEE_SUPPORT_DATATYPE 2851
IEEE_SUPPORT_DENORMAL 2852
IEEE_SUPPORT_DIVIDE 2852
IEEE_SUPPORT_FLAG 2853
IEEE_SUPPORT_HALTING 2854
IEEE_SUPPORT_INF 2854
IEEE_SUPPORT_IO 2855
IEEE_SUPPORT_NAN 2856
IEEE_SUPPORT_ROUNDING 2856
IEEE_SUPPORT_SQRT 2857
IEEE_SUPPORT_STANDARD 2858
IEEE_SUPPORT_UNDERFLOW_CONTROL 2859
IEEE_UNORDERED 2860
IEEE_VALUE 2860
intrinsic functions 1930, 1947, 1953, 2223, 2321, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2335, 2340, 2342, 2344, 2350, 2351, 2352, 2360, 2365, 2367, 2368, 2409, 2416, 2422, 2429, 2441, 2444, 2459, 2468, 2482, 2486, 2487, 2488, 2489, 2490, 2494, 2513, 2516, 2536, 2538, 2539, 2574, 2586, 2589, 2593, 2594, 2629, 2632, 2635, 2647, 2649, 2663, 2691, 2693, 2820, 2821, 2823, 2824, 2825, 2826, 2827, 2829, 2835, 2870, 2888, 2898, 2910, 2914, 2929, 2939, 2940, 2941, 2942, 2943, 2945, 2947, 2948, 2953, 2955, 2956, 2958, 2960, 2961, 2962, 2963, 2965, 2973, 2974, 2977, 2979, 2980, 2983, 2990, 2993, 2995, 2997, 2998, 3002, 3022, 3024, 3028, 3030, 3031, 3035, 3040, 3054, 3060, 3061, 3066, 3069, 3070, 3071, 3106, 3110, 3134, 3187, 3188, 3189, 3192, 3201, 3344, 3345, 3346, 3347, 3348, 3350, 3352, 3363, 3371, 3390, 3402, 3409, 3410, 3422, 3423, 3448, 3504, 3507, 3508, 3509, 3511, 3512, 3513,
intrinsic functions *(continued)* 3521, 3522, 3527, 3559, 3561, 3590, 3598, 3599, 3612, 3615, 3616, 3618, 3619, 3638, 3660
ABS 2321
ACHAR 2325
ACOS 2326
ACOSD 2327
ACOSH 2328
ADJUSTL 2328
ADJUSTR 2329
AIMAG 2330
AINT 2331
ALL 2335
ALLOCATED 2340
AND 2821
ANINT 2342
ANY 2344
ASIN 2350
ASIND 2351
ASINH 2352
ASSOCIATED 2360
ATAN 2365
ATAN2 2365
ATAN2D 2367
ATAND 2368
ATANH 2368
BADDRESS 2409
BIT_SIZE 2416
BTEST 2422
CACHESIZE 2429
categories of 1953
CEILING 2441
CHAR 2444
CMPLX 2459
COMMAND_ARGUMENT_COUNT 2468
CONJG 2482
COS 2486
COSD 2487
COSH 2488
COTAN 2488
COTAND 2489
COUNT 2490
CSHIFT 2494
DBLE 2513
DCMPLX 2516
DFLOAT 2536
DIGITS 2538
DIM 2539

intrinsic functions *(continued)*
DNUM 2574
DOT_PRODUCT 2586
DPROD 2589
DREAL 2593
DSHIFTL 2594
DSHIFTR 2594
EOF 2629
EOSHIFT 2632
EPSILON 2635
EXP 2647
EXPONENT 2649
FLOAT 3371
FLOOR 2663
for data representation models 2223
FP_CLASS 2691
FRACTION 2693
HUGE 2820
IACHAR 2820
IAND 2821
IARG 2823
IARGC 2823
IBCHNG 2824
IBCLR 2825
IBITS 2826
IBSET 2827
ICHAR 2829
IEEE_COPY_SIGN 2835
IEOR 2870
IFIX 2910
ILEN 2888
INDEX 2898
INT 2910
INT_PTR_KIND 2914
INUM 2929
IOR 2929
IS_IOSTAT_END 2939
IS_IOSTAT_EOR 2940
ISHA 2941
ISHC 2942
ISHFT 2943
ISHFTC 2945
ISHL 2947
ISNAN 2948
IXOR 2870
JNUM 2953
KIND 2955
KNUM 2956
LBOUND 2958

intrinsic functions *(continued)*
LEADZ 2960
LEN 2961
LEN_TRIM 2962
LGE 2963
LGT 2965
LLE 2973
LLT 2974
LOC 2977
LOG 2979
LOG10 2980
LOGICAL 2983
LSHFT 2943
LSHIFT 2943
MALLOC 2990
MATMUL 2993
MAX 2995
MAXEXPONENT 2997
MAXLOC 2998
MAXVAL 3002
MCLOCK 3022
MERGE 3024
MIN 3028
MINEXPONENT 3030
MINLOC 3031
MINVAL 3035
MOD 3040
MODULO 3054
MULT_HIGH_SIGNED (i64) 3061
MULT_HIGH (i64) 3060
NARGS 3066
NEAREST 3069
NEW_LINE 3070
NINT 3071
NOT 3106
NULL 3110
NUMARG 2823
OR 2929
PACK 3134
POPCNT 3187
POPPAR 3188
PRECISION 3189
PRESENT 3192
PRODUCT 3201
QCMPLX 3344
QEXT 3345
QFLOAT 3346
QNUM 3347
QREAL 3348

intrinsic functions *(continued)*
RADIX 3350
RAN 3352
RANGE 3363
REAL 3371
references to generic 1930
REPEAT 3390
RESHAPE 3390
RNUM 3402
RRSPACING 3402
RSHFT 2943
RSHIFT 2943
SCALE 3409
SCAN 3410
SELECTED_CHAR_KIND 3422
SELECTED_INT_KIND 3422
SELECTED_REAL_KIND 3423
SET_EXPONENT 3448
SHAPE 3504
SHIFTL 3507
SHIFTR 3508
SIGN 3509
SIN 3511
SIND 3512
SINH 3513
SIZE 3521
SIZEOF 3522
SNGL 3371
SPACING 3527
SPREAD 3559
SQRT 3561
SUM 3590
TAN 3598
TAND 3599
TANH 3599
TINY 3612
TRAILZ 3615
TRANSFER 3616
TRANSPOSE 3618
TRIM 3619
UNPACK 3638
VERIFY 3660
XOR 2870
intrinsic modules 1900, 1901, 1904, 1906
IEEE 1906
ISO_C_BINDING 1901
ISO_FORTRAN_ENV 1904
intrinsic procedures 1930, 1934, 1947, 1949, 2927
and EXTERNAL 1930
intrinsic procedures *(continued)*
and IMPLICIT 1930
argument keywords in 1949
classes of 1947
elemental 1947
nonelemental 1947
references to elemental 1934
references to generic 1930
scope of name 1930
using as actual arguments 2927
intrinsics 1602
intrinsic subroutine
IEEE_GET_FLAG 2835
IEEE_GET_HALTING_MODE 2836
IEEE_GET_ROUNDING_MODE 2837
IEEE_GET_STATUS 2838
IEEE_GET_UNDERFLOW_MODE 2839
IEEE_SET_ROUNDING_MODE 2848
IEEE_SET_STATUS 2849
IEEE_SET_UNDERFLOW_MODE 2850
intrinsic subroutines 1947, 1975, 2492, 2506, 2509, 2642, 2646, 2694, 2716, 2730, 2731, 2745, 2830, 3037, 3055, 3062, 3357, 3360, 3362, 3595, 3608
categories of 1975
CPU_TIME 2492
DATE 2506
DATE_AND_TIME 2509
ERRSNS 2642
EXIT 2646
FREE 2694
GET_COMMAND 2730
GET_COMMAND_ARGUMENT 2731
GET_ENVIRONMENT_VARIABLE 2745
GETARG 2716
IDATE 2830
MM_PREFETCH 3037
MOVE_ALLOC 3055
MVBITS 3062
RANDOM_NUMBER 3357
RANDOM_SEED 3360
RANDU 3362
SYSTEM_CLOCK 3595
TIME 3608
introduction to Building Applications 93
introduction to Compiler options 443
introduction to Optimizing Applications 1239
introduction to the Language Reference 1737
INUM 2929

inverse cosine
    function returning in degrees 2327
    function returning in radians 2326
inverse sine
    function returning in degrees 2351
    function returning in radians 2350
inverse tangent
    function returning in degrees 2368
    function returning in degrees (complex) 2365
    function returning in radians 2365
    function returning in radians (complex) 2367
invoking Intel(R) Fortran Compiler 95
IOFOCUS 2111, 2137
    specifier for INQUIRE 2111
    specifier for OPEN 2137
IOR 2929
IOSTAT 358, 361, 1985, 3365, 3673
    errors returned to 361
    specifier for READ 3365
    specifier for WRITE 3673
    symbolic definitions in iosdef.for 358
    using 358
IOSTAT specifier for CLOSE 2457
IPO
    options 1229
    option specifying jobs during the link phase of 702, 923
IPXFARGC 2931
IPXFCONST 2932
IPXFLENTRIM 2932
IPXFWEXITSTATUS 2933
IPXFWSTOPSIG 2935
IPXFWTERMSIG 2936
IQINT 2910
IQNINT 3071
IR 1497, 1501
IRAND 2936
IRANDM 2936
IRANGET 2938
IRANSET 2938
IS_IOSTAT_END 2939
IS_IOSTAT_EOR 2940
ISATTY 2939
ISHA 2941
ISHC 2942
ISHFT 2943
ISHFTC 2945
ISHL 2947
ISIGN 3509
ISNAN 2948
ISO_C_BINDING 1901
ISO_C_BINDING derived types 1901
ISO_C_BINDING intrinsic module 1901, 1904
    derived types 1901
    named constants 1901
    procedures 1904
ISO_C_BINDING named constants 1901
ISO_C_BINDING procedures 1904, 2424, 2425, 2426, 2427
    C_ASSOCIATED 2424
    C_F_POINTER 2425
    C_F_PROCPOINTER 2426
    C_FUNLOC 2427
    C_LOC 2427
ISO_FORTRAN_ENV 1904
ISO_FORTRAN_ENV intrinsic module 1904
iteration count 2575
iteration loop control 1886
iterative DO loops 1886
ITIME 2948
IVDEP 1474, 1581, 2949
    effect of compiler option on 1474
    effect when tuning applications 1581
IVDEP directive 1596
IXOR 2870
IZEXT 3676

J

JABS 2951
Japan Industry Standard characters 3011
JDATE 2952
JDATE4 2953
JFIX 2910
JIAND 2821
JIBCLR 2825
JIBITS 2826
JIBSET 2827
JIDIM 2539
JIDINT 2910
JIDNNT 3071
JIEOR 2870
JIFIX 2910
JINT 2910
JIOR 2929
JIQINT 2910

JIS characters
converting to JMS 3011
JISHFT 2943
JISHFTC 2945
JISIGN 3509
JIXOR 2870
JMAX0 2995
JMAX1 2995
JMIN0 3028
JMIN1 3028
JMOD 3040
JMS characters
converting to JIS 3011
JMVBITS 3062
JNINT 3071
JNOT 3106
JNUM 2953
jump tables
option enabling generation of 776, 947
JZEXT 3676

K

KDIM 2539
KEEP value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
keyboard character
function returning ASCII value of 2892
keyboard procedures
table of 2264
key code charts 2211
key codes 2216, 2217, 2218
chart 1 2217
chart 2 2218
keystroke
function checking for 3158
function returning next 2723
keywords 1750
KIABS 2321
KIAND 2821
KIBCLR 2825
KIBITS 2826
KIBSET 2827
KIDIM 2539
KIDINT 2910
KIDNNT 3071
KIEOR 2870
KIFIX 2910
KILL 2954, 3279
POSIX version of 3279
KIND 2916, 2955, 3370
directive specifying default for integers 2916
directive specifying default for reals 3370
kind type parameter 174, 176, 1763, 1770, 2955, 2983, 3422, 3423, 3626
declaring for data 3626
function changing logical 2983
function returning for character data 3422
function returning for integer data 3422
function returning for real data 3423
function returning value of 2955
INTEGER declarations 174
LOGICAL declarations 176
restriction for real constants 1770
KINT 2910
KIOR 2929
KIQINT 2910
KIQNNT 3071
KISHFT 2943
KISHFTC 2945
KISIGN 3509
KMAX0 2995
KMAX1 2995
KMIN0 3028
KMIN1 3028
KMOD 3040
KMP_AFFINITY 1383, 1418
modifier 1418
offset 1418
permute 1418
type 1418
KMP_ALL_THREADS 1383
KMP_BLOCKTIME 1383
KMP_LIBRARY 1383, 1406
KMP_MONITOR_STACKSIZE 1383
KMP_STACKSIZE 1383
KMP_VERSION 1383
KMVBITS 3062
KNINT 3071
KNOT 3106
KNUM 2956
KZEXT 3676

L

L 2061
  edit descriptor 2061
label assignment 2352
labels 81, 1752, 2352, 2575, 2810, 2811, 2813, 2873
  assigning 2352
  general rules for 1752
  in DO constructs 2575
  platform 81
  statement transferring control to 2813
  statement transferring control to assigned 2810
  statement transferring control to one of three 2873
  statement transferring control to specified 2811
language and country combinations
  function returning array of 3073
language compatibility 1741
language extensions
  and portability 429
  built-in functions 2236
  character sets 2234
  compilation control statements 2236
  compiler directives 2239
  convention for 81
  C Strings 2235
  data in expressions 2235
  directive enabling or disabling Intel Fortran 3103, 3577
  dollar sign () allowed in names 2234
  file operation statements 2237
  for execution control 2235
  for source forms 2233
  general directives 2239
  Hollerith constants 2235
  i/o formatting 2237
  i/o statements 2236
  Intel Fortran 2233
  intrinsic procedures 2240
  language features for compatibility 2244
  number of characters in names 2234
  run-time library routines 2245
  specification statements 2235
  summary of 2233
  syntax for intrinsic data types 2234
language features for compatibility 2244
Language Reference
  overview 1737
language standards 429, 1741
  and portability 429
language standards *(continued)*
  conformance 1741
language summary tables 2248
LASTPRIVATE 1342, 1346, 2579, 2957, 3145, 3146, 3418
  in DO directive 2579
  in PARALLEL DO directive 3145
  in PARALLEL SECTIONS directive 3146
  in SECTIONS directive 3418
  summary of data scope attribute clauses 1342
  using 1346
LBOUND 1860, 2958
  in pointer assignment 1860
LCWRQQ 2959
LD_LIBRARY_PATH environment variable 129
LEADZ 2960
left shift
  function performing arithmetic 2941
  function performing circular 2942
  function performing logical 2947
LEN 1779, 1849, 2961
  in CHARACTER data type 1779
  in declaration statements 1849
LEN_TRIM 2962
LEN= 1779, 1849
  in CHARACTER data type 1779
  in declaration statements 1849
length
  specifying for character objects 1849
length specifier in character declarations 1849
lexical string comparisons
  function determining 2974
  function determining 2973
  function determining > 2965
  function determining > or = 2963
LGE 2963
LGT 2965
libgcc library
  option linking dynamically 1060
  option linking statically 1069
libraries 97, 319, 321, 325, 329, 504, 508, 747, 828, 1065, 1360, 1398, 1402, 1630, 1638, 1643
  creating shared 321
  default 97
  IFPORT portability module 325
  libifcore.lib 1638
  math 329
  OpenMP* run-time routines 1398, 1402
  option enabling dynamic linking of 504

libraries *(continued)*
option enabling static linking of 508
option preventing linking with shared 1065
option preventing use of standard 747
option printing location of system 828
static 319
libraries used when linking 331
library
option searching in specified directory for 709
option to search for 708
library directory paths when linking 331
library exception handler
overriding 411
library functions 1360, 1398, 1402, 1511
Intel extension 1402
OpenMP* run-time routines 1398
to control number of threads 1360
library math functions
option testing errno after calls to 594
library routines 323, 2148, 2229, 2230, 2247, 2248
how to use 323
module 2229
OpenMP Fortran 2230
using to open files 2148
library search path
directive placing in file 3114
license file
specifying the location of 129
limitations of mixed-language programming 264
limits
Intel(R) Visual Fortran compiler 438
line length
directive setting for fixed-source format 2657
lines
function drawing 2966
function drawing between arrays 2968, 2970
function drawing within an array 3185
line style
function returning 2772
subroutine setting 3462
LINETO 2966
LINETO_W 2966
LINETOAR 2968
LINETOAREX 2970
linker
option passing linker option relax to 738
option passing linker option to 1118
option passing options to 715, 1110
option telling to read commands from file 1071
linker *(continued)*
option to prevent running 109
request threaded run-time library 101
using from the command line 107
viewing libraries used 331
linker diagnostic messages 331
linker error conditions 331
linker library directory paths 331
linker options for search libraries
option including in object files 710
linking
option preventing use of startup files and libraries when 751
option preventing use of startup files when 750
option suppressing 109, 509
linking options 1229
linking tools 1229, 1497, 1504, 1508
xild 1497, 1504, 1508
xilibtool 1508
xilink 1497, 1504
linking with IPO 1501
link map file
generating 103
option generating 720
Linux* compiler options
-1 764, 935
-132 565
-66 570
-72 565
-80 565
-align 472
-allow fpp_comments 476
-altparam 478
-ansi-alias 479, 847
-arch 480
-assume 486
-auto 498
-auto_scalar 496, 848
-autodouble 1046
-automatic 498
-ax 500, 850
-B 503
-Bdynamic 504
-Bstatic 508
-c 509
-C 511
-CB 511
-ccdefault 510
-check 511

Linux* compiler options *(continued)*
-cm 1099
-common-args 486
-complex-limited-range 516, 855
-convert 517
-cpp 625, 889
-CU 511
-cxxlib 520
-cxxlib-gcc 520
-cxxlib-nostd 520
-D 522
-DD 523, 856
-debug 526
-debug-parameters 532
-diag 533, 539, 857, 863
-diag-dump 538, 862
-diag-enable sc-include 544, 867
-diag-enable sc-parallel 545, 869
-diag-error-limit 547, 871
-diag-file 548, 872
-diag-file-append 550, 873
-diag-id-numbers 551, 875
-diag-once 552, 876
-d-lines 523, 856
-double-size 554
-dps 478
-dryrun 556
-dynamic-linker 558
-dyncom 560, 877
-E 561
-e03 1099
-e90 1099
-e95 1099
-EP 562
-error-limit 547, 871
-extend-source 565
-f66 570
-f77rtl 572
-falias 573
-falign-functions 574, 882
-falign-stack 575
-fast 577
-fast-transcendentals 578, 879
-fcode-asm 580
-fexceptions 581
-ffnalias 582
-FI 588
-finline 583
-finline-functions 584

Linux* compiler options *(continued)*
-finline-limit 585
-finstrument-functions 586, 912
-fixed 588
-fkeep-static-consts 589, 928
-fltconsistency 590
-fma 593, 880
-fmath-errno 594
-fminshared 596
-fnsplit 597, 883
-fomit-frame-pointer 598, 600, 803
-fp 598, 600, 803
-fpconstant 616
-fpe 617
-fpe-all 620
-fpic 623
-fpie 624
-fp-model 601, 606
-fpp 625, 889
-fp-port 611, 884
-fp-relaxed 612, 885
-fpscomp 627
-fp-speculation 613, 886
-fp-stack-check 615, 888
-FR 638
-fr32 637
-free 638
-fsource-asm 639
-fstack-protector 640, 641, 661
-fstack-security-check 640, 641, 661
-fsyntax-only 1070
-ftrapuv 642, 1024
-ftz 643, 891
-funroll-loops 1026, 1085
-fverbose-asm 646
-fvisibility 647
-g 650, 1119, 1122
-gdwarf-2 655
-gen-interfaces 657
-global-hoist 658, 893
-heap-arrays 662
-help 663
-I 667
-i2 691
-i4 691
-i8 691
-idirafter 669
-i-dynamic 1058
-inline-debug-info 676, 896

Linux* compiler options *(continued)*
-inline-factor 677, 898
-inline-forceinline 679, 900
-inline-level 680, 758
-inline-max-per-compile 682, 901
-inline-max-per-routine 683, 903
-inline-max-size 685, 905
-inline-max-total-size 687, 906
-inline-min-size 688, 908
-intconstant 690
-integer-size 691
-ip 693, 914
-IPF-fltacc 698, 919
-IPF-flt-eval-method0 696, 917
-IPF-fma 593, 880
-IPF-fp-relaxed 612, 885
-ip-no-inlining 694, 915
-ip-no-pinlining 695, 916
-ipo 699, 920
-ipo-c 701, 922
-ipo-jobs 702, 923
-ipo-S 704, 925
-ipo-separate 705, 926
-i-static 1068
-isystem 706
-ivdep-parallel 707, 927
-l 708
-L 709
-logo 716
-lowercase 744
-m 717
-m32 719
-m64 719
-map-opts 721, 931
-march 723
-mcmodel 724
-mcpu 740
-mieee-fp 590
-minstruction 730, 911
-mixed-str-len-arg 670
-mkl 732, 933
-module 734
-mp 590
-mp1 737, 989
-mrelax 738
-mtune 740
-multiple-processes 735, 743
-names 744
-nbs 486

Linux* compiler options *(continued)*
-no-bss-init 746, 934
-nodefaultlibs 747
-nofor_main 748
-nolib-inline 749
-nostartfiles 750
-nostdinc 1116
-nostdlib 751
-nus 486
-o 752
-O 753
-Ob 680, 758
-onetrip 764, 935
-openmp 765, 936
-openmp-lib 766, 937
-openmp-link 768, 939
-openmp-profile 770, 940
-openmp-report 771, 942
-openmp-stubs 772, 943
-openmp-threadprivate 774, 944
-opt-block-factor 775, 946
-opt-jump-tables 776, 947
-opt-loadpair 778, 948
-opt-malloc-options 779
-opt-mem-bandwidth 780, 949
-opt-mod-versioning 782, 951
-opt-multi-version-aggressive 783, 952
-opt-prefetch 784, 953
-opt-prefetch-initial-values 786, 955
-opt-prefetch-issue-excl-hint 787, 956
-opt-prefetch-next-iteration 788, 957
-opt-ra-region-strategy 790, 959
-opt-report 791, 960
-opt-report-file 793, 962
-opt-report-help 794, 963
-opt-report-phase 795, 964
-opt-report-routine 796, 965
-opt-streaming-stores 797, 966
-opt-subscript-in-range 799, 968
-Os 800
-p 805
-P 827
-pad 806, 971
-pad-source 807, 972
-par-affinity 808, 975
-parallel 819, 986
-par-num-threads 810, 977
-par-report 811, 978
-par-schedule 814, 980

Linux* compiler options *(continued)*
-par-threshold 818, 984
-pc 821, 987
-pg 805
-pie 823
-prec-div 825, 990
-prec-sqrt 826, 991
-preprocess-only 827
-print-multi-lib 828
-prof-data-order 829, 992
-prof-dir 830, 994
-prof-file 832, 995
-prof-func-groups 833
-prof-func-order 834, 996
-prof-gen 836, 998
-prof-genx 836, 998
-prof-hotness-threshold 838, 1000
-prof-src-dir 840, 1001
-prof-src-root 841, 1003
-prof-src-root-cwd 843, 1005
-prof-use 845, 1006
-Qinstall 910
-Qlocation 929
-Qoption 969
-qp 805
-r16 1046
-r8 1046
-rcd 1008, 1044
-rct 1009, 1045
-real-size 1046
-recursive 1047
-reentrancy 1049
-RTCu 511
-S 1050
-safe-cray-ptr 1010, 1051
-save 1012, 1053
-save-temps 1013, 1054
-scalar-rep 1015, 1056
-shared 1057
-shared-intel 1058
-shared-libgcc 1060
-sox 1017, 1062
-stand 1063
-static 1065
-static-intel 1068
-static-libgcc 1069
-std 1063
-std03 1063
-std90 1063
Linux* compiler options *(continued)*
-std95 1063
-syntax-only 1070
-T 1071
-tcheck 1019, 1072
-tcollect 1020, 1073
-tcollect-filter 1021, 1075
-Tf 1061
-threads 1077
-tprofile 1023, 1078
-traceback 1080
-tune 1081
-u 1099
-U 1084
-unroll 1026, 1085
-unroll-aggressive 1027, 1086
-uppercase 744
-us 486
-use-asm 1028, 1088
-v 1089
-V 1090
-vec 1032, 1090
-vec-guard-write 1033, 1091
-vec-report 1034, 1092
-vec-threshold 1036, 1094
-vms 1095
-w 1098, 1099
-W0 1099
-W1 1099
-Wa 1098
-warn 1099
-watch 1105
-WB 1106
-what 1107
-Wl 1110
-Wp 1111
-x 1038, 1112
-X 1116
-Xlinker 1118
-y 1070
-zero 1042, 1121
-Zp 472, 1124
list-directed formatting
input 2000
output 2019
list-directed i/o
default formats for output 2019
input 2000
output 2019

list-directed i/o *(continued)*
restrictions for input 2000
list-directed I/O 208
list-directed input 2000
list-directed output 2019
list-directed statements
READ 2000
WRITE 2019
list-drected I/O statements 203
list items in i/o lists 1991
literal constants 1763
LITTLE_ENDIAN 2130
value for CONVERT specifier 2130
little-endian-to-big-endian conversion 1664
LLE 2973
LLT 2974
LNBLNK 2975
LOADIMAGE 2976
LOADIMAGE_W 2976
loadpair optimization
option enabling 778, 948
LOC 2977, 3166
using with integer pointers 3166
locale
function returning currency string for current 3074
function returning date for current 3076
function returning information about current 3083
function returning number string for current 3078
function returning time for current 3079
local scope 2171
local variables
option allocating to static memory 1012, 1053
option allocating to the run-time stack 498
locating run-time errors
using traceback information 412
locations
specifying alternative 99
lock routines 1398
LOG 2979
LOG10 2980
logarithm
function returning base 10 2980
function returning common 2980
function returning natural 2979
logarithmic procedures
table of 2269
LOGICAL 2982, 2983
LOGICAL(1) 1778
LOGICAL(2) 1778
LOGICAL(4) 1778
LOGICAL(8) 1778
LOGICAL*1 1778
LOGICAL*2 1778
LOGICAL*4 1778
LOGICAL*8 1778
logical AND
function performing 2821
logical assignment statements 1836
logical complement
function returning 3106
logical constants 1779
logical conversion
function performing 2983
logical data
mixed-language programming 290
logical data representation 176
logical data type 176, 187, 290, 1778, 1779, 2186
constants 1779
converting nonnative data 187
declaring 176
default kind 1778
differences with nonnative formats 187
mixed-language programming 290
ranges 176
representation 176
storage 2186
logical devices 197
logical editing (L) 2061
logical expressions 1825, 2875
conditional execution based on value of 2875
evaluating 1825
logical IF statement 2875
logical operations 1825
data types resulting from 1825
logical operators 1825
logical records 208
logical shift
function performing 2943
function performing left 2943
function performing right 2943
logical unit number
function testing whether it's a terminal 2939
logical units
assigning files to 205
logical values
transferring 2061
login name
subroutine returning 2774

LONG 2983
loop blocking factor
option specifying 775, 946
loop control 1886, 2575
DO WHILE 1886
iterative 1886
simple 1886
LOOP COUNT 1591, 2984
and loop distribution 1591
loop directives
DISTRIBUTE POINT 2544
general rules for 2162
IVDEP 2949
LOOP COUNT 2984
PARALLEL and NOPARALLEL 3142, 3143
PREFETCH and NOPREFETCH 3100, 3189
SWP and NOSWP (i64) 3105, 3592
UNROLL_AND_JAM and NOUNROLL_AND_JAM 3644
UNROLL and NOUNROLL 3108, 3643
VECTOR ALIGNED and VECTOR UNALIGNED 3654, 3659
VECTOR ALWAYS and NOVECTOR 3109, 3655
VECTOR NONTEMPORAL (i32, i64em) 3657, 3658
VECTOR TEMPORAL (i32, i64em) 3657, 3658
loop interchange 1638
loops 775, 946, 1026, 1027, 1085, 1086, 1242, 1453, 1462, 1466, 1581, 1583, 1584, 1591, 1595, 1611, 1623, 1630, 1638, 1647, 1658, 1886, 1888, 2498, 2575, 2645, 2876, 3108, 3643, 3644
anti dependency 1584
collapsing 1630
constructs 1466
controlling number of times unrolled 3108, 3643
count 1591, 1595
dependencies 1453
distribution 1581, 1591
DO 2575
DO WHILE 1886
enabling jamming 3644
flow dependency 1584
IF 2876
independence 1584
interchange 1581, 1611, 1638, 1647
iterative 1886
limiting loop unrolling 3108, 3643
manual transformation 1647
nested DO 1888
option specifying blocking factor for 775, 946
loops *(continued)*
option specifying maximum times to unroll 1026, 1085
option using aggressive unrolling for 1027, 1086
output dependency 1584
parallelization 1242, 1453, 1462
reductions 1584
simple 1886
skipping DO 2498
terminating DO 2645
transformations 1581, 1638, 1658
unrolling 1583, 1595
using for arrays 1623
vectorization 1462
loop unrolling 1260, 1461, 1581, 1583, 1595
limitations of 1583
support for 1595
using the HLO optimizer 1260, 1581
lower bounds
function returning 2958
lowercase names
case-sensitivity 274, 275, 276
LSHFT 2943
LSHIFT 2943
LSTAT 2986
LST files 103
LTIME 2987

M

machine epsilon 1727
Mac OS* X compiler options
-1 764, 935
-132 565
-66 570
-72 565
-80 565
-align 472
-allow fpp_comments 476
-altparam 478
-ansi-alias 479, 847
-arch 480
-assume 486
-auto 498
-auto_scalar 496, 848
-autodouble 1046
-automatic 498
-ax 500, 850

Mac OS* X compiler options *(continued)*
-B 503
-c 509
-C 511
-CB 511
-ccdefault 510
-check 511
-cm 1099
-common-args 486
-complex-limited-range 516, 855
-convert 517
-cpp 625, 889
-CU 511
-cxxlib 520
-cxxlib-gcc 520
-cxxlib-nostd 520
-D 522
-DD 523, 856
-debug 526
-debug-parameters 532
-diag 533, 539, 857, 863
-diag-dump 538, 862
-diag-enable sc-include 544, 867
-diag-enable sc-parallel 545, 869
-diag-error-limit 547, 871
-diag-file 548, 872
-diag-file-append 550, 873
-diag-id-numbers 551, 875
-diag-once 552, 876
-d-lines 523, 856
-double-size 554
-dps 478
-dryrun 556
-dynamiclib 559
-dyncom 560, 877
-E 561
-e03 1099
-e90 1099
-e95 1099
-EP 562
-error-limit 547, 871
-extend-source 565
-f66 570
-f77rtl 572
-falias 573
-falign-functions 574, 882
-falign-stack 575
-fast 577
-fast-transcendentals 578, 879

Mac OS* X compiler options *(continued)*
-fcode-asm 580
-fexceptions 581
-ffnalias 582
-FI 588
-finline 583
-finline-functions 584
-finline-limit 585
-finstrument-functions 586, 912
-fixed 588
-fkeep-static-consts 589, 928
-fltconsistency 590
-fmath-errno 594
-fminshared 596
-fomit-frame-pointer 598, 600, 803
-fp 598, 600, 803
-fpconstant 616
-fpe 617
-fpe-all 620
-fpic 623
-fp-model 601, 606
-fpp 625, 889
-fp-port 611, 884
-fpscomp 627
-fp-speculation 613, 886
-fp-stack-check 615, 888
-FR 638
-free 638
-fsource-asm 639
-fstack-protector 640, 641, 661
-fstack-security-check 640, 641, 661
-fsyntax-only 1070
-ftrapuv 642, 1024
-ftz 643, 891
-funroll-loops 1026, 1085
-fverbose-asm 646
-fvisibility 647
-g 650, 1119, 1122
-gdwarf-2 655
-gen-interfaces 657
-global-hoist 658, 893
-heap-arrays 662
-help 663
-I 667
-i2 691
-i4 691
-i8 691
-idirafter 669
-i-dynamic 1058

Mac OS* X compiler options *(continued)*
-inline-factor 677, 898
-inline-forceinline 679, 900
-inline-level 680, 758
-inline-max-per-compile 682, 901
-inline-max-per-routine 683, 903
-inline-max-size 685, 905
-inline-max-total-size 687, 906
-inline-min-size 688, 908
-intconstant 690
-integer-size 691
-ip 693, 914
-ip-no-inlining 694, 915
-ip-no-pinlining 695, 916
-ipo 699, 920
-ipo-c 701, 922
-ipo-S 704, 925
-ipo-separate 705, 926
-i-static 1068
-isystem 706
-l 708
-L 709
-logo 716
-lowercase 744
-m 717
-m32 719
-m64 719
-map-opts 721, 931
-march=pentium4 723
-mcpu 740
-mdynamic-no-pic 729
-mieee-fp 590
-minstruction 730, 911
-mixed_str_len_arg 670
-mkl 732, 933
-module 734
-mp 590
-mp1 737, 989
-mtune 740
-multiple-processes 735, 743
-names 744
-nbs 486
-no-bss-init 746, 934
-nodefaultlibs 747
-nofor-main 748
-nolib-inline 749
-nostartfiles 750
-nostdinc 1116
-nostdlib 751

Mac OS* X compiler options *(continued)*
-nus 486
-o 752
-O 753
-Ob 680, 758
-onetrip 764, 935
-openmp 765, 936
-openmp-link 768, 939
-openmp-report 771, 942
-openmp-stubs 772, 943
-opt-block-factor 775, 946
-opt-jump-tables 776, 947
-opt-malloc-options 779
-opt-multi-version-aggressive 783, 952
-opt-ra-region-strategy 790, 959
-opt-report 791, 960
-opt-report-file 793, 962
-opt-report-help 794, 963
-opt-report-phase 795, 964
-opt-report-routine 796, 965
-opt-streaming-stores 797, 966
-opt-subscript-in-range 799, 968
-p 805
-P 827
-pad 806, 971
-pad-source 807, 972
-parallel 819, 986
-par-num-threads 810, 977
-par-report 811, 978
-par-schedule 814, 980
-par-threshold 818, 984
-pc 821, 987
-pg 805
-prec-div 825, 990
-prec-sqrt 826, 991
-preprocess-only 827
-print-multi-lib 828
-prof-data-order 829, 992
-prof-dir 830, 994
-prof-file 832, 995
-prof-func-groups 833
-prof-func-order 834, 996
-prof-gen 836, 998
-prof-genx 836, 998
-prof-src-dir 840, 1001
-prof-src-root 841, 1003
-prof-src-root-cwd 843, 1005
-prof-use 845, 1006
-Qinstall 910

Mac OS* X compiler options *(continued)*
-Qlocation 929
-Qoption 969
-qp 805
-r16 1046
-r8 1046
-rcd 1008, 1044
-rct 1009, 1045
-real-size 1046
-recursive 1047
-reentrancy 1049
-RTCu 511
-S 1050
-safe-cray-ptr 1010, 1051
-save 1012, 1053
-save-temps 1013, 1054
-scalar-rep 1015, 1056
-shared-intel 1058
-shared-libgcc 1060
-stand 1063
-static-intel 1068
-staticlib 1066
-static-libgcc 1069
-std 1063
-std03 1063
-std90 1063
-std95 1063
-syntax-only 1070
-Tf 1061
-threads 1077
-traceback 1080
-tune 1081
-u 1099
-U 1084
-unroll 1026, 1085
-unroll-aggressive 1027, 1086
-uppercase 744
-us 486
-use-asm 1028, 1088
-v 1089
-V 1090
-vec 1032, 1090
-vec-guard-write 1033, 1091
-vec-report 1034, 1092
-vec-threshold 1036, 1094
-vms 1095
-w 1098, 1099
-W1 1099
-Wa 1098
Mac OS* X compiler options *(continued)*
-warn 1099
-watch 1105
-WB 1106
-what 1107
-Wl 1110
-Wp 1111
-x 1038, 1112
-X 1116
-Xlinker 1118
-y 1070
-zero 1042, 1121
-Zp 472, 1124
main program 1897, 1898, 2603, 3203
statement identifying 3203
statement terminating 2603
maintainability 1344, 1402, 1474, 1611, 1636, 1696
access 1611
allocation 1402
copying data in 1344
dependency 1474
layout 1611
main thread
option adjusting the stack size for 974
make command 114
MAKEDIRQQ 2989
makefiles
command-line syntax 114
MALLOC 2990, 3166
using with integer pointers 3166
mantissa in real model 2225
manual transformations 1647
many-one array section 1810, 1838
MAP 103, 2623, 3634
files 103
MASK 1949
masked array assignment 2682, 3666
generalization of 2682
mask expressions
function combining arrays using 3024
function counting true elements using 2490
function determining all true using 2335
function determining any true using 2344
function finding location of maximum value using 2998
function finding location of minimum value using 3031
function packing array using 3134

mask expressions *(continued)*
function returning maximum value of elements using 3002
function returning minimum value of elements using 3035
function returning product of elements using 3201
function returning sum of elements using 3590
function unpacking array using 3638
in ELSEWHERE 3666
in FORALL 2682
in WHERE 3666
mask pattern
subroutine setting newone for fill 3451
MASTER 2992
master thread
copying data in 2485
specifying code to be executed by 2992
math functions
option enabling faster code sequences for 612, 885
math libraries 329
MATMUL 2993
matrix multiplication
function performing 2993
MAX 2995
MAX0 2995
MAX1 2995
MAXEXPONENT 2997
maximum exponent
function returning 2997
maximum value
function returning 2995
function returning location of 2998
maximum value of array elements
function returning 3002
MAXLOC 2998
MAXREC 2138
MAXVAL 3002
MBCharLen 3004
MBConvertMBToUnicode 3005
MBConvertUnicodeToMB 3007
MBCS routines
table of 2301
MBCurMax 3008
MBINCHARQQ 3009
MBINDEX 3010
MBJISToJMS 3011
MBJMSToJIS 3011
MBLead 3012
MBLen 3013
MBLen_Trim 3014
MBLEQ 3015
MBLGE 3015
MBLGT 3015
MBLLE 3015
MBLLT 3015
MBLNE 3015
MBNext 3017
MBPrev 3018
MBSCAN 3019
MBStrLead 3020
MBVERIFY 3021
MCLOCK 3022
memory
dynamically allocating 2338
freeing space associated with allocatable arrays 2517
freeing space associated with pointer targets 2517
function allocating 2990
subroutine freeing allocated 2694
using EQUIVALENCE to share 1863
memory aliasing 1638
memory allocation procedures
table of 2264
memory bandwidth
option enabling tuning and heuristics for 780, 949
memory cache
function returning size of a level in 2429
memory deallocation procedures
table of 2264
memory dependency
option specifying no loop-carried following IVDEP 707, 927
memory file system 1630
memory layout
option changing variable and array 806, 971
memory loads
option enabling optimizations to move 658, 893
memory location
directive updating dynamically 3022
memory model
option specifying large 724
option specifying small or medium 724
option to use specific 724
memory space
deallocating 2517
MEMORYTOUCH 3022
MEMREF_CONTROL 3023

menu command
    function simulating selection of 2455
menu items
    function changing callback routine of 3043
    function changing text string of 3045
    function deleting 2527
    function inserting 2907
    function modifying the state of 3042
menus
    function appending child window list to 3497
    function appending item to 2345
    function inserting item in 2907
    function setting top-level for append list 3497
menu state
    constants indicating 2345, 2907, 3042
MERGE 3024
MESSAGE 3026
message box
    function displaying 3026
    function specifying text for About 2320
MESSAGEBOXQQ 3026
messages
    display of run-time 353
    meaning of severity to run-time system 353
    run-time error 361
    run-time format 353
methods of specifying the data format 188
Microsoft* Fortran PowerStation
    option specifying compatibility with 627
Microsoft* Visual C++
    option specifying compatibility with 1031
Microsoft* Visual Studio
    option specifying compatibility with 1031
Microsoft Fortran PowerStation
    compatibility with 246
    compatible file types 246
midnight
    function returning seconds since 3416
MIN 3028
MIN0 3028
MIN1 3028
MINEXPONENT 3030
minimum exponent
    function returning 3030
minimum value
    function returning 3028
    function returning location of 3031
minimum value of array elements
    function returning 3035
MINLOC 3031
MINVAL 3035
misaligned data 1475
miscellaneous run-time procedures
    table of 2317
MIXED_STR_LEN_ARG 2386, 2389
    option for ATTRIBUTES directive 2386, 2389
mixed language programming
    calling subprograms 263
mixed-language programming 264, 266, 274, 275, 276, 280, 281, 283, 285, 289, 290, 293, 294, 301, 305, 306
    adjusting case of names 276
    adjusting naming conventions in 274, 275
    allocatable arrays in 293
    array pointers in 293
    calling conventions in 266
    exchanging and accessing data in 280
    handling data types in 289
    integer pointers in 294
    limitations 264
    naming conventions 274
    numeric data types in 290
    overview of issues 264
    passing arguments in 281
    return values 290, 301
    summary of issues 264
    user-defined types in 305
    using common external data in 285
    using modules in 283
mixed-language programs 268, 274, 275, 278, 279, 311, 312
    compiling and linking 311
mixed-language projects
    programming with 263
mixed-mode expressions 1821
mixing vectorizable types in a loop 1461, 1623
    using effectively 1623
MM_PREFETCH 3037
MMX(TM) 1459
mock object files 1501
MOD 3040
MODE 2112, 2138
    specifier for INQUIRE 2112
    specifier for OPEN 2138
model
    for bit data 2227
    for integer data 2224
    for real data 2225

models for data representation 2223, 2224, 2225, 2227
  bit 2227
  integer 2224
  real 2225
MODIFYMENUFLAGSQQ 3042
MODIFYMENUROUTINEQQ 3043
MODIFYMENUSTRINGQQ 3045
MODULE 3047
module entities 2183, 3205
  attribute limiting use of 3205
module files
  option specifying directory for 734
module naming conventions 278
MODULE PROCEDURE 3053
module procedures 1897, 1898, 1918, 2923, 3047, 3053
  in interface blocks 2923
  in modules 3047
  internal procedures in 1918
module references 1899
modules 258, 283, 325, 1745, 1897, 1898, 1899, 3047, 3196, 3208, 3645
  accessibility of entities in 3196, 3208, 3645
  advantages of 258
  allowing access to 3645
  common blocks in 1898
  defining 3047
  IFPORT 325
  overview of 1745
  private entities in 3196
  public entities in 3208
  references to 1899
  use in mixed-language programming 283
  USE statement in 3645
module subprograms 1898, 2483
  CONTAINS statement 2483
MODULO 3054
modulo computation
  function returning 3054
modulo operations
  option enabling versioning of 782, 951
mouse cursor
  function setting the shape of 3466
mouse events
  function registering callback routine for 3383
  function unregistering callback routine for 3641
  function waiting for 3664
mouse input
  function waiting for 3664
MOVBE instructions
  option generating 730, 911
MOVE_ALLOC 3055
MOVETO 3058
MOVETO_W 3058
MULT_HIGH 3060
MULT_HIGH_SIGNED 3061
multibyte characters 107, 3008, 3009, 3010, 3012, 3015, 3019, 3020, 3021
  function performing context-sensitive test for 3020
  function returning first 3012
  function returning length for codepage 3008
  function returning number and character 3009
  functions comparing strings of 3015
  incharqq function for 3009
  index function for 3010
  scan function for 3019
  verify function for 3021
multibyte-character string
  function converting to codepage 3007
  function converting to Unicode 3005
  function returning length (including blanks) 3013
  function returning length (no blanks) 3014
  function returning length of first character in 3004
  function returning position of next character in 3017
  function returning position of previous character in 3018
multidimensional arrays
  construction of 1812, 3390
  conversion between vectors and 3134, 3638
  storage of 1804
multiple processes
  option creating 735, 743
  routines for working with 1495
multiple processes vs multiple threads 1495
multithread 1479
multithread applications 101, 118, 1049, 1479, 1480, 1481, 1488, 1491, 1492
  compiling and linking 101
  creating 1479
  critical applications 1488
  errors in 1492
  events 1488
  handling errors in 1492
  modules for 1481
  mutex 1488

multithread applications *(continued)*
option generating reentrant code for 1049
overview of 1479
programming considerations 1480
routines for 1492
semaphore 1488
sharing data 1480
sharing resources 1488
suspending a thread 1491
synchronizing threads 1488, 1491
table of 1492
thread local storage 1491
writing 1480
multithreaded programs 168, 1242, 1447, 1611
debugging 168
multithreading 1406, 1453, 1630, 1658, 1671
data 1658, 1671
records 1671
storage 1630
multithreading concepts 1479
multi-threading performance
option aiding analysis of 1023, 1078
multithreads 1479
MVBITS 3062
MXCSR register 1689

N

NAME 2112, 2138
specifier for INQUIRE 2112
specifier for OPEN 2138
name association 2183, 2185, 2186
argument 2183
pointer 2185
storage 2186
NAMED 2112
named common 2417, 2473
defining initial values for variables in 2417
named constants 1763, 3148
NAMELIST 3064
namelist external records
alternative form for 2200
namelist formatting 1985, 2003, 2021
input 2003
output 2021
namelist group 2003, 2021, 3064
prompting for information about 2003
namelist I/O 208, 2003, 2021
input 2003
output 2021
namelist input 2003
comments in 2003
namelist output 2021
namelist records 2003
namelists 3064
namelist specifier 1985
namelist statements
READ 2003
WRITE 2021
names 1748, 1799, 2171, 2175, 2180, 2181, 2889, 3064, 3148, 3203
associating with constant value 3148
associating with group 3064
association of 2181
explicit typing of 1799
first character in 1748
in PARAMETER statements 3148
length allowed 1748
of main programs 3203
overriding default data typing of 2889
resolving references to nonestablished 2180
scope of 2171
statement defining default types for user-defined 2889
unambiguous 2175
naming conventions
mixed-language programming 274
NaN values
function testing for 2948
NARGS 3066
NATIVE 2130
value for CONVERT specifier 2130
native and nonnative numeric formats 181
natural alignment 1613
NEAREST 3069
nearest different number
function returning 3069
nearest integer
function returning 3071
nested and group repeat specifications 2086
nested DO constructs 1888
nested IF constructs 2876
NEW_LINE 3070
new line character
function returning 3070
NEXTREC 2113

NINT 3071
NLS date and time format 3083
NLSEnumCodepages 3072
NLSEnumLocales 3073
NLSFormatCurrency 3074
NLSFormatDate 3076
NLSFormatNumber 3078
NLSFormatTime 3079
NLS functions
    date and time format 3083
    MBCharLen 3004
    MBConvertMBToUnicode 3005
    MBConvertUnicodeToMB 3007
    MBCurMax 3008
    MBINCHARQQ 3009
    MBINDEX 3010
    MBJISToJMS and MBJMSToJIS 3011
    MBLead 3012
    MBLen 3013
    MBLen_Trim 3014
    MBLEQ 3015
    MBLGE 3015
    MBLGT 3015
    MBLLE 3015
    MBLLT 3015
    MBLNE 3015
    MBNext 3017
    MBPrev 3018
    MBSCAN 3019
    MBStrLead 3020
    MBVERIFY 3021
    NLSEnumCodepages 3072
    NLSEnumLocales 3073
    NLSFormatCurrency 3074
    NLSFormatDate 3076
    NLSFormatNumber 3078
    NLSFormatTime 3079
    NLSGetEnvironmentCodepage 3081
    NLSGetLocale 3082
    NLSGetLocaleInfo 3083
    NLSSetEnvironmentCodepage 3095
    NLSSetLocale 3096
    table of 2301
NLSGetEnvironmentCodepage 3081
NLSGetLocale 3082
NLSGetLocaleInfo 3083
NLS language
    function setting current 3096
    subroutine retrieving current 3082
NLS locale parameters
    table of 3083
NLS parameters
    table of 3083
NLSSetEnvironmentCodepage 3095
NLSSetLocale 3096
nmake command 114
NML 1985, 3365, 3673
    specifier for READ 3365
    specifier for WRITE 3673
NO_ARG_CHECK 2387
    option for ATTRIBUTES directive 2387
NODECLARE 140, 2518
    equivalent compiler option for 140
NOFREEFORM 140, 2695, 3098
    equivalent compiler option for 140
NOINLINE 2384, 2385, 2388
    option for ATTRIBUTES directive 2384, 2385, 2388
NOMIXED_STR_LEN_ARG 2386, 2389
    option for ATTRIBUTES directive 2386, 2389
nonadvancing i/o 1988
nonadvancing I/O 208
nonadvancing record I/O 238
nonblock DO 2575
    terminal statements for 2575
noncharacter data types 1847
noncharacter type declaration statements 1847
nondecimal numeric constants 1792, 1795
    determining the data type of 1795
nonelemental functions 1947
nonexecutable statements 1746
non-Fortran procedures
    references to 1935
    referencing with %LOC 2978
nonnative data
    porting 187
nonrepeatable edit descriptors 2031, 2068
non-unit memory access 1638
NOOPTIMIZE 140, 3099, 3123
    equivalent compiler option for 140
NOPARALLEL 3142, 3143
NOPREFETCH 1602, 3100, 3189
    using 1602
normalized floating-point number 1722
NOSHARED 2138
NOSTRICT 140, 3103, 3577
    equivalent compiler option for 140
NOSWP 1605, 3105, 3592
    using 1605

NOT 3106
Not-a-Number (NaN) 1722, 2948
  function testing for 2948
NOUNROLL 3108, 3643
NOUNROLL_AND_JAM 3644
NOVECTOR 3109, 3655
NOVECTOR directive 1596
NOWAIT 2579, 2664, 3378, 3418, 3520
  effect on implied FLUSH directive 2664
  effect with REDUCTION clause 3378
  in END DO directive 2579
  in END SECTIONS directive 3418
  in END SINGLE directive 3520
NUL 2345
  predefined QuickWin routine 2345
NULL 3110
NULLIFY 1875, 3112
  overview of dynamic allocation 1875
NUM_THREADS 3139, 3145
  in PARALLEL directive 3139
  in PARALLEL DO directive 3145
NUMARG 2823
NUMBER 2113
number string
  function returning for current locale 3078
numeric assignment statements 1834
numeric constants
  complex 1775
  integer 1766
  nondecimal 1792
  real 1769
numeric conversion
  limitations of 1721
numeric data
  size limits for A editing 2065
numeric data types
  conversion rules with DATA 2500
  mixed-language programming 290
numeric expressions 1818, 1820, 1821, 1823
  comparing values of 1823
  data type of 1821
  using parentheses in 1820
numeric format
  specifying 2042, 2130
  specifying with /convert 196
  specifying with OPEN(CONVERT=) 194
  specifying with OPTIONS statement 195
numeric functions
  categories of 1953
numeric functions *(continued)*
  models defining 2223
numeric models 2223, 2224, 2225, 2227, 2820, 3612
  bit 2227
  integer 2224
  querying parameters in 2820, 3612
  real 2225
numeric nondecimal constants 1792, 1795
  determining the data type of 1795
numeric operators 1818
  precedence of 1818
numeric parameters
  functions returning 1725
  retrieving parameters of 1725
numeric procedures
  table of 2267
numeric routines 1953
numeric storage unit 2186

O

O 2046
  edit descriptor 2046
OBJCOMMENT 140, 3114
  equivalent compiler option for 140
object file
  directive specifying library search path 3114
  option generating one per source file 705, 926
  option increasing number of sections in 506
  option placing a text string into 507
  option specifying name for 760
object module
  directive specifying identifier for 2833
obj files 1658
OBJ files 121
octal constants 1792, 1793, 2199
  alternative syntax for 2199
octal editing (O) 2046
octal values
  transferring 2046
of allocatable arrays 1878
of pointer targets 1880
OMP_DYNAMIC 1383
OMP_NESTED 1383
OMP_NUM_THREADS 1383
OMP_SCHEDULE 1383

OMP directives 1242, 1299, 1321, 1326, 1352, 1359, 1360, 1364, 1370, 1371, 1375, 1379, 1383, 1398, 1406, 1418
advanced issues 1375
compatibility libraries 1321, 1406
compatibility with other compilers 1321
debugging 1375
directives 1359, 1360, 1364, 1370, 1371
environment variables 1383, 1418
guidelines for using libraries 1321
KMP_AFFINITY 1383, 1418
KMP_ALL_THREADS 1383
KMP_BLOCKTIME 1383
KMP_LIBRARY 1383
KMP_MONITOR_STACKSIZE 1383
KMP_STACKSIZE 1383
KMP_VERSION 1383
legacy libraries 1321, 1406
library file names 1406
object-level interoperability 1321
OMP_DYNAMIC 1383
OMP_NESTED 1383
OMP_NUM_THREADS 1383
OMP_SCHEDULE 1383
omp.h 1375
parallel processing thread model 1326
performance 1375
reports 1299
run-time library routines 1398
source compatibility 1321
support libraries 1406
ONLY 3645
keyword in USE statement 3645
OPEN 181, 187, 194, 213, 358, 2120, 2125, 2126, 2127, 2128, 2129, 2130, 2133, 2134, 2136, 2137, 2138, 2139, 2140, 2141, 2143, 2144, 2146, 2147, 2148, 3115
ACCESS specifier 2125
ACTION specifier 2125
ASSOCIATEVARIABLE specifier 2126
BLANK specifier 2127
BLOCKSIZE specifier 2127
BUFFERCOUNT specifier 2128
BUFFERED specifier 2128
CARRIAGECONTROL specifier 2129
CONVERT specifier 181, 194, 2130
DEFAULTFILE specifier 2133
defaults for converting nonnative data 187
DELIM specifier 2133
OPEN *(continued)*
DISPOSE specifier 2134
example of ERR specifier 358
example of FILE specifier 358
example of IOSTAT specifier 358
FILE specifier 2134
FORM specifier 2136
IOFOCUS specifier 2137
MAXREC specifier 2138
MODE specifier 2138
NAME specifier 2138
NOSHARED specifier 2138
ORGANIZATION specifier 2138
PAD specifier 2139
POSITION specifier 2139
READONLY specifier 2140
RECL specifier
option to specify units 213
units for unformatted files 187
RECORDSIZE specifier 2143
RECORDTYPE specifier 2143
SHARED specifier 2146
SHARE specifier 2144
STATUS specifier 2146
table of specifiers and values 2120
TITLE specifier 2147
TYPE specifier 2148
USEROPEN specifier 2148
OPENED 2113
specifier for INQUIRE 2113
opening files 228, 3115
OPEN statement 228
OpenMP*
option controlling diagnostics 771, 942
option enabling 765, 936
option enabling analysis of applications 770, 940
option enabling programs in sequential mode 772, 943
option specifying threadprivate 774, 944
OpenMP* Fortran directives 1352, 1364, 1379, 1392, 1402, 2159, 2164, 2166, 2167, 2168, 2253, 2369, 2409, 2492, 2579, 2664, 2992, 3129, 3139, 3145, 3146, 3147, 3418, 3520, 3602, 3606, 3607, 3670
ATOMIC 2369
BARRIER 2409
clauses for 1392, 2166
conditional compilation of 2167
CRITICAL 2492

OpenMP* Fortran directives *(continued)*
DO 2579
examples of 1379
features of 1352
FLUSH 2664
for synchronization 1364
for worksharing 1352
Intel extensions for 1402
MASTER 2992
nesting and binding rules 2168
ORDERED 3129
PARALLEL 3139
PARALLEL DO 3145
PARALLEL SECTIONS 3146
PARALLEL WORKSHARE 3147
programming using 1352
SECTION 3418
SECTIONS 3418
SINGLE 3520
syntax of 2159
table of 2253
TASK 3602
TASKWAIT 3606
THREADPRIVATE 3607
WORKSHARE 3670
OpenMP* run-time library
option controlling which is linked to 768, 939
option specifying 766, 937
OpenMP* run-time library routines 2230
OpenProcess 1495
OPEN statement 228
operands 1817, 1818, 1825
in logical expressions 1825
in numeric expressions 1818
operating system
portability considerations 432
operations
character 1823
complex 1821
conversion to higher precision 1821
defined 1827
integer 1821
numeric 1818
real 1821
operator precedence
summary of 1827
operators 1817, 1818, 1823, 1825, 1827, 1940
binary 1818
generic 1940
operators *(continued)*
logical 1825
numeric 1818
precedence of 1827
relational 1823
unary 1818
optimal records to improve performance 1245, 1247, 1251, 1260, 1302, 1314, 1591, 1605, 1630
analyzing applications 1251
application-specific 1251
hardware-related 1251
library-related 1251
methodology 1247
options
restricting 1314
setting 1314
OS-related 1251
reports 1260, 1591, 1605
resources 1245
restricting 1314
setting 1314
strategies 1251
system-related 1251
optimization
compiling and linking for 100
controlling unrolling and jamming 3644
directive affecting 3099, 3123
limiting loop unrolling 3108, 3643
loop unrolling 3108, 3643
option disabling all 753
option enabling global 763
option enabling prefetch insertion 784, 953
option generating single assembly file from multiple files 704, 925
option generating single object file from multiple files 701, 922
option specifying code 753
preventing with VOLATILE 3661
specified by ATOMIC directive 2369
specified by SWP and NOSWP directives (i64) 3105, 3592
specified by UNROLL_AND_JAM and NOUNROLL_AND_JAM directives 3644
specified by UNROLL and NOUNROLL directives 3108, 3643
optimization report
option displaying phases for 794, 963
option generating for routines with specified text 796, 965

optimization report *(continued)*
option generating to stderr 791, 960
option specifying detail level of 791, 960
option specifying name for 793, 962
option specifying phase to use for 795, 964
optimizations 761, 800, 802, 1239, 1242, 1301, 1302, 1306, 1313, 1314, 1519, 1520, 1581, 1591, 1658
compilation process 1301
default level of 1658
for specific processors 1306
high-level language 1581
option disabling all 761
option enabling all speed 802
option enabling many speed 800
options for IA-32 architecture 1306
options for IA-64 architecture 1313
options for Intel(R) 64 architecture 1306
overview of 1301, 1519
parallelization 1242
PGO methodology 1520
profile-guided 1519
profile-guided optimization 1520
support features for 1591
optimization support 1239
OPTIMIZE 140, 3099, 3123
equivalent compiler option for 140
optimizer report generation 1260
optimizing 1239, 1251, 1638
applications 1251
helping the compiler 1638
overview 1239
technical applications 1251
optimizing performance 1247
OPTIONAL 3118
optional arguments 1923, 3118, 3192
function determining presence of 3192
optional plus sign in output fields 2071
option mapping tool 138
options
compiler 137
precedence using CONVERT 194
specifying unformatted file floating-point format 195
OPTIONS 3122, 3124
options for efficient compilation 1658
options for IA-32 architectures 1310
options used for IPO 1229, 1500
OptReport support 1260
OR 2929
ORDERED 1364, 1371, 1392, 2579, 3129, 3145
clause in DO directive 2579
clause in PARALLEL DO directive 3145
example of 1364
in DO directives 1371
overview of OpenMP* directives and clauses 1392
order of subscript progression 1804
ORGANIZATION 2114, 2138
specifier for INQUIRE 2114
specifier for OPEN 2138
OUTGTEXT 2764, 2766, 3130, 3455, 3460
related routines 2764, 2766, 3455, 3460
output
displaying to screen 3194
output files
option specifying name for 752
producing 122
output statements for data transfer
PRINT 3194
REWRITE 3398
WRITE 3673
OUTTEXT 3133, 3671
effect of WRAPON 3671
overflow 1360, 1398, 1595, 1605, 1660
call to a runtime library routine 1398
loop unrolling 1595
software pipelining 1605
the threads number 1360
overriding vectorization 1596
overview 157, 168, 325, 1239, 1242, 1302, 1342, 1658
configuration files 157
IFPORT portability module 325
of data scope attribute clauses 1342
of debugging multithreaded programs 168
of optimizing compilation 1658
of optimizing for specific processors 1302
of parallelism 1242
portability library 325
response files 157
overview of Building Applications 93

P

P 2074
edit descriptor 2074

PACK 140, 3134, 3136
    equivalent compiler option for 140
packed array
    function creating 3134
packed structures 1642
PACKTIMEQQ 3138
PAD 2114, 2139
    specifier for INQUIRE 2114
    specifier for OPEN 2139
padding for blanks 208
padding short source lines
    for fixed and tab source 1757
    for free source 1754
page keys
    function determining behavior of 3150
PARALLEL 1351, 2164, 3139, 3142, 3143
    general directive 3142, 3143
    OpenMP* Fortran directive 2164, 3139
    using SHARED clause in 1351
parallel compiler directives 2164
parallel construct 1352
PARALLEL DO 1342, 1344, 1345, 1346, 1348, 1351, 1359, 1364, 1392, 3145
    and synchronization constructs 1364
    example of 1359
    SCHEDULE clause 1342
    summary of OpenMP* directives and clauses 1392
    using COPYIN clause in 1344
    using DEFAULT clause in 1345
    using FIRSTPRIVATE clause in 1346
    using LASTPRIVATE clause in 1346
    using PRIVATE clause in 1346
    using REDUCTION clause in 1348
    using SHARED clause in 1351
parallel invocations with makefile 1520
parallelism 1242, 1398, 1447
parallelization 545, 869, 1242, 1287, 1447, 1453, 1462
    diagnostic 1287
    option enabling analysis in source code 545, 869
parallel library routines 2230
parallel lint 1331
PARALLEL OpenMP* directive 1344, 1345, 1346, 1348, 1351, 1364, 1392
parallel processing 1326, 1352
    thread model 1326
parallel programming 1239, 1242
parallel region
    directive defining 3139
parallel region *(continued)*
    option specifying number of threads to use in 810, 977
parallel regions 1360, 1392
    directive defining 1360
    directives affecting 1360
    library routine affecting 1360
PARALLEL SECTIONS 1344, 1345, 1346, 1348, 1351, 1359, 1364, 1392, 3146
    and synchronization constructs 1364
    example of 1359
    summary of OpenMP* directives 1392
    using COPYIN clause in 1344
    using DEFAULT clause in 1345
    using FIRSTPRIVATE clause in 1346
    using LASTPRIVATE clause in 1346
    using PRIVATE clause in 1346
    using REDUCTION clause in 1348
    using SHARED clause in 1351
PARALLEL WORKSHARE 1359, 3147
    using 1359
PARAMETER 478, 3148
    option allowing alternative syntax 478
parentheses
    effect in character expressions 1823
    effect in logical expressions 1825
    effect in numeric expressions 1820, 1821
partial association 2186
PASSDIRKEYSQQ 3150
passing 1623
    array arguments efficiently 1623
passing by reference
    %REF 3381
path
    function splitting into components 3528
PATH environment variable 129
pathnames
    specifying default 227
pattern used to fill shapes
    subroutine returning 2759
PAUSE 3156
PEEKCHARQQ 3158
PENDING
    specifier for INQUIRE 2114
performance 1696
performance analyzer 1643
performance issues with IPO 1504
PERROR 3159
PGO 1519

PGO API
    _PGOPTI_Prof_Dump_And_Reset 1579
    _PGOPTI_Prof_Reset 1579
    _PGOPTI_Set_Interval_Prof_Dump 1578
    enable 1573
pgopti.dpi file 1520, 1574
pgopti.spi file 1552
PGO tools 1532, 1552, 1561
    code coverage tool 1532
    profmerge 1561
    proforder 1561
    test prioritization tool 1552
pgouser.h header file 1573
physical coordinates
    subroutine converting from viewport coordinates 2775
    subroutine converting to viewport coordinates 2798
PIE 3160
PIE_W 3160
pie graphic
    function testing for endpoints of 2714
pie-shaped wedge
    function drawing 3160
pipelining 1242, 1591, 1605
    affect of LOOP COUNT on 1591
    for IA-64 architecture based applications 1605
pixel
    function returning color index for 2777
    function returning RGB color value for 2779
    function setting color index for 3469
    function setting RGB color value for 3470
pixels
    function returning color index for multiple 2781
    function returning RGB color value for multiple 2782
    function setting color index for multiple 3473
    function setting RGB color value for multiple 3474
platform labels 81
POINTER 1840, 1875, 3112, 3163, 3166
    attribute 1840, 1875, 3112
    integer 3166
pointer aliasing 783, 952, 1660
    option using aggressive multi-versioning check for 783, 952
pointer arguments 1925, 1937
    requiring explicit interface 1937
pointer assignment 1840
pointer association function 2360
pointer association status 1925
pointers
    allocating 2338
    assigning values to targets of 1833, 2357
    assignment of 1840
    associating with targets 1840, 3600
    CRAY-style 3166
    disassociating 2517
    disassociating from targets 3112
    dynamic association of 1875
    Fortran 95/90 3163
    function retuning association status of 2360
    function returning disassociated 3110
    initial association status of 3112
    initializing 3110
    integer 3166
    nullifying 3112
    option checking for disassociated 511
    option checking for uninitialized 511
    referencing 3163
    volatile 3661
    when storage space is created for 1875
pointer targets 1875, 1877, 1880, 2338, 2517, 3600
    allocation of 1877
    as dynamic objects 1875
    creating 2338
    deallocation of 1880
    freeing memory associated with 2517
POLYBEZIER 3169
POLYBEZIER_W 3169
POLYBEZIERTO 3175
POLYBEZIERTO_W 3175
POLYGON 3181
POLYGON_W 3181
polygons
    function drawing 3181
POLYLINEQQ 3185
POPCNT 3187
POPPAR 3188
portability considerations
    and data representation 432
    and the operating system 432
    data transportability 433
    overview 429
    recommendations 430
    using IFPORT portability module 325
portability library
    overview 325
portability routines 326, 430, 2288, 2319, 2324, 2333, 2410, 2411, 2412, 2416, 2420, 2440,

portability routines *(continued)*
2442, 2443, 2446, 2449, 2452, 2456, 2457, 2479, 2497, 2507, 2508, 2511, 2515, 2525, 2528, 2537, 2590, 2592, 2595, 2643, 2652, 2653, 2656, 2666, 2692, 2696, 2697, 2702, 2703, 2722, 2732, 2737, 2738, 2740, 2741, 2744, 2745, 2748, 2754, 2764, 2769, 2770, 2774, 2777, 2785, 2795, 2796, 2808, 2819, 2831, 2832, 2833, 2861, 2867, 2871, 2887, 2901, 2913, 2936, 2938, 2939, 2948, 2951, 2952, 2953, 2954, 2959, 2975, 2983, 2986, 2987, 2989, 3138, 3215, 3347, 3348, 3351, 3353, 3355, 3363, 3364, 3387, 3388, 3401, 3403, 3404, 3412, 3415, 3417, 3420, 3438, 3441, 3442, 3444, 3448, 3450, 3485, 3508, 3513, 3516, 3523, 3524, 3525, 3528, 3530, 3531, 3533, 3535, 3536, 3537, 3540, 3542, 3543, 3544, 3545, 3546, 3548, 3549, 3550, 3553, 3554, 3556, 3557, 3558, 3562, 3564, 3593, 3596, 3609, 3611, 3620, 3637, 3640
ABORT 2319
ACCESS 2324
ALARM 2333
BEEPQQ 2410
BESJN 2411
BESYN 2411
BIC 2412
BIS 2412
BIT 2416
BSEARCHQQ 2420
CDFLOAT 2440
CHANGEDIRQQ 2442
CHANGEDRIVEQQ 2443
CHDIR 2446
CHMOD 2449
CLEARSTATUSFPQQ 2452
CLOCK 2456
CLOCKX 2457
COMPLINT 2479
COMPLLOG 2479
COMPLREAL 2479
CSMG 2497
CTIME 2497
DATE 2507
DATE4 2508
DBESJN 2511
DBESYN 2511
DCLOCK 2515
DELDIRQQ 2525

portability routines *(continued)*
DELFILESQQ 2528
DFLOATI 2537
DFLOATJ 2537
DFLOATK 2537
DRAND 2590
DRANDM 2590
DRANSET 2592
DTIME 2595
ETIME 2643
FDATE 2652
FGETC 2653
FINDFILEQQ 2656
FLUSH 2666
FPUTC 2692
FSEEK 2696
FSTAT 2697
FTELL 2702
FTELLI8 2702
FULLPATHQQ 2703
GETC 2722
GETCONTROLFPQQ 2732
GETCWD 2737
GETDAT 2738
GETDRIVEDIRQQ 2740
GETDRIVESIZEQQ 2741
GETDRIVESQQ 2744
GETENV 2745
GETENVQQ 2748
GETFILEINFOQQ 2754
GETGID 2764
GETLASTERROR 2769
GETLASTERRORQQ 2770
GETLOG 2774
GETPID 2777
GETPOS 2785
GETPOSI8 2785
GETSTATUSFPQQ 2785
GETTIM 2795
GETTIMEOFDAY 2796
GETUID 2796
GMTIME 2808
HOSTNAM 2819
IDATE 2831
IDATE4 2832
IDFLOAT 2833
IEEE_FLAGS 2861
IEEE_HANDLER 2867
IERRNO 2871

portability routines *(continued)*
IFLOATI 2887
IFLOATJ 2887
INMAX 2901
INTC 2913
IRAND and IRANDM 2936
IRANGET 2938
IRANSET 2938
ISATTY 2939
ITIME 2948
JABS 2951
JDATE 2952
JDATE4 2953
KILL 2954
LCWRQQ 2959
LNBLNK 2975
LONG 2983
LSTAT 2986
LTIME 2987
MAKEDIRQQ 2989
PACKTIMEQQ 3138
PUTC 3215
QRANSET 3347
QSORT 3348
RAISEQQ 3351
RAND 3353
RANDOM function 3353
RANDOM subroutine 3355
RANF 3363
RANGET 3364
RANSET 3364
recommendations 430
RENAME 3387
RENAMEFILEQQ 3388
RINDEX 3401
RTC 3403
RUNQQ 3404
SCANENV 3412
SCWRQQ 3415
SECNDS 3417
SEED 3420
SETCONTROLFPQQ 3438
SETDAT 3441
SETENVQQ 3442
SETERRORMODEQQ 3444
SETFILEACCESSQQ 3448
SETFILETIMEQQ 3450
SETTIM 3485
SHORT 3508

portability routines *(continued)*
SIGNAL 3513
SIGNALQQ 3516
SLEEP 3523
SLEEPQQ 3524
SORTQQ 3525
SPLITPATHQQ 3528
SPORT_CANCEL_IO 3530
SPORT_CONNECT 3531
SPORT_CONNECT_EX 3533
SPORT_GET_HANDLE 3535
SPORT_GET_STATE 3536
SPORT_GET_STATE_EX 3537
SPORT_GET_TIMEOUTS 3540
SPORT_PEEK_DATA 3542
SPORT_PEEK_LINE 3543
SPORT_PURGE 3544
SPORT_READ_DATA 3545
SPORT_READ_LINE 3546
SPORT_RELEASE 3548
SPORT_SET_STATE 3549
SPORT_SET_STATE_EX 3550
SPORT_SET_TIMEOUTS 3553
SPORT_SHOW_STATE 3554
SPORT_SPECIAL_FUNC 3556
SPORT_WRITE_DATA 3557
SPORT_WRITE_LINE 3558
SRAND 3562
SSWRQQ 3564
STAT 3564
SYSTEM 3593
SYSTEMQQ 3596
table of 2288
TIME 3609
TIMEF 3611
TTYNAM 3620
UNLINK 3637
UNPACKTIMEQQ 3640
POS 1989, 2115, 3365, 3673
specifier for INQUIRE 2115
specifier for READ 3365
specifier for WRITE 3673
POSITION 2116, 2139
specifier for INQUIRE 2116
specifier for OPEN 2139
positional editing 2070, 2071
T 2070
TL 2071
TR 2071

positional editing *(continued)*
X 2071
position-independent code
option generating 623, 624
position-independent executable
option producing 823
position-independent external references
option generating code with 729
position of file
functions returning 2702, 2785
specifying 2139
POSIX* routines 2304, 2931, 2932, 2933, 2935, 2936, 3220, 3221, 3223, 3224, 3226, 3227, 3228, 3229, 3230, 3231, 3232, 3233, 3234, 3235, 3236, 3237, 3238, 3239, 3241, 3242, 3243, 3244, 3245, 3248, 3250, 3251, 3252, 3254, 3256, 3257, 3258, 3259, 3260, 3261, 3262, 3263, 3264, 3265, 3268, 3269, 3271, 3272, 3273, 3274, 3275, 3276, 3277, 3278, 3279, 3280, 3281, 3282, 3283, 3284, 3288, 3289, 3291, 3292, 3293, 3294, 3295, 3296, 3297, 3299, 3300, 3301, 3302, 3303, 3304, 3305, 3306, 3307, 3308, 3309, 3310, 3311, 3312, 3318, 3319, 3322, 3323, 3324, 3325, 3326, 3327, 3328, 3329, 3333, 3334, 3335, 3336, 3338, 3340, 3342, 3343
IPXFARGC 2931
IPXFCONST 2932
IPXFLENTRIM 2932
IPXFWEXITSTATUS (L*X, M*X) 2933
IPXFWSTOPSIG (L*X, M*X) 2935
IPXFWTERMSIG (L*X, M*X) 2936
PXF(type)GET 3220
PXF(type)SET 3221
PXFA(type)SET 3224
PXFACCESS 3226
PXFACHARGET 3223
PXFACHARSET 3224
PXFADBLGET 3223
PXFADBLSET 3224
PXFAGET 3223
PXFAINT8GET 3223
PXFAINT8SET 3224
PXFAINTGET 3223
PXFAINTSET 3224
PXFALARM 3227
PXFALGCLGET 3223
PXFALGCLSET 3224
PXFAREALGET 3223
POSIX* routines *(continued)*
PXFAREALSET 3224
PXFASTRGET 3223
PXFASTRSET 3224
PXFCALLSUBHANDLE 3228
PXFCFGETISPEED (L*X, M*X) 3229
PXFCFGETOSPEED (L*X, M*X) 3229
PXFCFSETISPEED (L*X, M*X) 3230
PXFCFSETOSPEED (L*X, M*X) 3231
PXFCHARGET 3220
PXFCHARSET 3221
PXFCHDIR 3231
PXFCHMOD 3232
PXFCHOWN (L*X, M*X) 3233
PXFCLEARENV 3233
PXFCLOSE 3234
PXFCLOSEDIR 3234
PXFCONST 3235
PXFCREAT 3236
PXFCTERMID 3237
PXFDBLGET 3220
PXFDBLSET 3221
PXFDUP 3237
PXFDUP2 3237
PXFE(type)GET 3238
PXFE(type)SET 3239
PXFECHARGET 3238
PXFECHARSET 3239
PXFEDBLGET 3238
PXFEDBLSET 3239
PXFEINT8GET 3238
PXFEINT8SET 3239
PXFEINTGET 3238
PXFEINTSET 3239
PXFELGCLGET 3238
PXFELGCLSET 3239
PXFEREALGET 3238
PXFEREALSET 3239
PXFESTRGET 3238
PXFESTRSET 3239
PXFEXECV 3241
PXFEXECVE 3242
PXFEXECVP 3243
PXFEXIT 3244
PXFFASTEXIT 3244
PXFFCNTL (L*X, M*X) 3245
PXFFDOPEN 3248
PXFFFLUSH 3250
PXFFGETC 3250

POSIX* routines *(continued)*
PXFFILENO 3251
PXFFORK (L*X, M*X) 3252
PXFFPATHCONF 3254
PXFFPUTC 3256
PXFFSEEK 3257
PXFFSTAT 3258
PXFFTELL 3258
PXFGETARG 3259
PXFGETATTY 3260
PXFGETC 3260
PXFGETCWD 3261
PXFGETEGID (L*X, M*X) 3261
PXFGETENV 3262
PXFGETEUID (L*X, M*X) 3263
PXFGETGID (L*X, M*X) 3263
PXFGETGRGID (L*X, M*X) 3264
PXFGETGRNAM (L*X, M*X) 3264
PXFGETGROUPS (L*X, M*X) 3265
PXFGETLOGIN 3268
PXFGETPGRP (L*X, M*X) 3269
PXFGETPID 3269
PXFGETPPID 3271
PXFGETPWNAM (L*X, M*X) 3272
PXFGETPWUID (L*X, M*X) 3273
PXFGETSUBHANDLE 3274
PXFGETUID (L*X, M*X) 3275
PXFINT8GET 3220
PXFINT8SET 3221
PXFINTGET 3220
PXFINTSET 3221
PXFISBLK 3275
PXFISCHR 3276
PXFISCONST 3276
PXFISDIR 3277
PXFISFIFO 3278
PXFISREG 3278
PXFKILL 3279
PXFLGCLGET 3220
PXFLGCLSET 3221
PXFLINK 3280
PXFLOCALTIME 3281
PXFLSEEK 3282
PXFMKDIR 3283
PXFMKFIFO (L*X, M*X) 3284
PXFOPEN 3284
PXFOPENDIR 3288
PXFPATHCONF 3289
PXFPAUSE 3291

POSIX* routines *(continued)*
PXFPIPE 3292
PXFPOSIXIO 3292
PXFPUTC 3293
PXFREAD 3294
PXFREADDIR 3295
PXFREALGET 3220
PXFREALSET 3221
PXFRENAME 3295
PXFREWINDDIR 3296
PXFRMDIR 3297
PXFSETENV 3297
PXFSETGID (L*X, M*X) 3299
PXFSETPGID (L*X, M*X) 3300
PXFSETSID (L*X, M*X) 3301
PXFSETUID (L*X, M*X) 3301
PXFSIGACTION 3302
PXFSIGADDSET (L*X, M*X) 3303
PXFSIGDELSET (L*X, M*X) 3304
PXFSIGEMPTYSET (L*X, M*X) 3305
PXFSIGFILLSET (L*X, M*X) 3306
PXFSIGISMEMBER (L*X, M*X) 3306
PXFSIGPENDING (L*X, M*X) 3307
PXFSIGPROCMASK (L*X, M*X) 3308
PXFSIGSUSPEND (L*X, M*X) 3309
PXFSLEEP 3310
PXFSTAT 3310
PXFSTRGET 3220
PXFSTRSET 3221
PXFSTRUCTCOPY 3311
PXFSTRUCTCREATE 3312
PXFSTRUCTFREE 3318
PXFSYSCONF 3319
PXFTCDRAIN (L*X, M*X) 3322
PXFTCFLOW (L*X, M*X) 3322
PXFTCFLUSH (L*X, M*X) 3323
PXFTCGETATTR (L*X, M*X) 3324
PXFTCGETPGRP (L*X, M*X) 3325
PXFTCSENDBREAK (L*X, M*X) 3326
PXFTCSETATTR (L*X, M*X) 3326
PXFTCSETPGRP (L*X, M*X) 3327
PXFTIME 3328
PXFTIMES 3329
PXFTTYNAM (L*X, M*X) 3333
PXFUCOMPARE 3333
PXFUMASK 3334
PXFUNAME 3334
PXFUNLINK 3335
PXFUTIME 3335

POSIX* routines *(continued)*
PXFWAIT (L*X, M*X) 3336
PXFWAITPID (L*X, M*X) 3338
PXFWIFEXITED (L*X, M*X) 3340
PXFWIFSIGNALED (L*X, M*X) 3342
PXFWIFSTOPPED (L*X, M*X) 3342
PXFWRITE 3343
table of 2304
PRECISION 3189
precision in real model 2225, 3189
function querying 3189
preconnected units 197
predefined QuickWin routines 2345, 2907, 3043
preempting functions 1511
PREFETCH 1589, 1602, 3100, 3189
options used for 1589
using 1602
prefetches before a loop
option enabling 786, 955
prefetches for memory access in next iteration
option enabling 788, 957
prefetches of data 1589, 1602, 3037, 3100, 3189
directive enabling 3100, 3189
optimizations for 1589
subroutine peforming 3037
prefetch insertion
option enabling 784, 953
preparing code 1352
preprocessing directives
fpp 146
preprocessor
fpp 143
preprocessor definitions
option undefining all previous 1083
option undefining for a symbol 1083
preprocessor symbols
predefined 153
PRESENT 3192
pretested DO 2584
PRINT 3194
PRINT/DELETE value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
printing of formatted records 2089
printing to the screen 3194
PRINT value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
prioritizing application tests 1552, 1611
achieving optimum performance for 1611
PRIVATE 1342, 1345, 1346, 1348, 1360, 1371, 1392, 2521, 2579, 3139, 3145, 3146, 3196, 3200, 3418, 3520
in DEFAULT clause 1345, 2521
in DO directive 2579
in PARALLEL directive 3139
in PARALLEL DO directive 3145
in PARALLEL SECTIONS directive 3146
in SECTIONS directive 3418
in SINGLE directive 3520
in the DO directive 1371
relationship to REDUCTION clause 1348
summary of data scope attribute clauses 1342, 1392
used in PARALLEL directive 1360
using 1346
private entities 3196, 3645
procedure interface 1897, 1935, 1937, 1938, 1940, 1942, 2923
defining generic assignment 1942
defining generic names 1938
defining generic operators 1940
when explicit is required 1937
procedure names 275
procedure references 2175, 2176, 2179, 2180
resolving generic 2176
resolving nonestablished 2180
resolving specific 2179
unambiguous generic 2175
procedures
BLOCK DATA 2417
declaring external 2650
declaring intrinsic 2927
defining generic assignment for 1942
defining generic names for 1938
defining generic operators for 1940
directive specifying properties of 2371
dummy 1929
elemental user-defined 2596
external 1745, 1918
function computing address of 2978
generic 1938
interface blocks for mixed-language programming 279
interfaces to 1935, 2923
internal 1745, 1918
intrinsic 1947
mixed-language programming 279
module 1898, 2923, 3047, 3053

procedures *(continued)*
overview of intrinsic 1947
preventing side effects in 3212
pure user-defined 3212
recursive 3377
references to generic 1930
references to non-Fortran 1935
requiring explicit interface 1937
resolving references to 2175
specifying explicit interface for 2923
specifying intrinsic 2927
table of i/o 2252
procedures that require explicit interfaces 1937
process
function executing a new 3404
function returning ID of 2777
function returning user ID of 2796
processes
working with multiple 1495
processes and threads 1479
concepts for multithreaded applications 1479
processes and threads/concepts for multithreaded applications 1479
process execution
subroutine suspending 3523
process ID
function returning 2777
function sending signal to 2954
processor
option optimizing for specific 651, 653, 740
processor clock
subroutine returning data from 3595
processors
targeting IA-32 architecture processors using options 1306
targeting IA-64 architecture processors using options 1313
targeting Intel(R) 64 architecture processors using options 1306
processor-specific code
option generating 500, 850
option generating and optimizing 1038, 1112
processor time
function returning 2515
subroutine returning 2492
PRODUCT 3201
product of array elements
function returning 3201
PROF_DIR environment variable 1574
PROF_DUMP_INTERVAL environment variable (deprecated) 1574
PROF_NO_CLOBBER environment variable 1574
profile data records
option affecting search for 840, 1001
option letting you use relative paths when searching for 841, 843, 1003, 1005
profile-guided optimization 1519, 1520, 1530, 1566, 1573, 1574, 1578, 1579
API support 1573
data ordering optimization 1566
dumping profile information 1579
environment variables 1574
example of 1530
function grouping optimization 1566
function ordering optimization 1566
function order lists optimization 1566
interval profile dumping 1578
options 1520
overview 1519
phases 1530
resetting dynamic profile counters 1579
resetting profile information 1579
support 1573
usage model 1519
profile-optimized code 1520, 1573, 1576, 1578, 1579
dumping 1576, 1578
generating information 1573
resetting dynamic counters for 1579
profiling
option enabling use of information from 845, 1006
option instrumenting a program for 836, 998
option specifying directory for output files 830, 994
option specifying name for summary 832, 995
profiling information
option enabling function ordering 834, 996
option using to order static data items 829, 992
profmerge 1561
profmerge.exe file 437
proforder.exe file 437
PROGRAM 3203
program control
transferring to CASE construct 2433
program control procedures
table of 2258
program control statements
table of 2258
program execution
statement suspending 3156

program execution *(continued)*
stopping 3575
subroutine delaying 3524
subroutine terminating 2646
program loops 1326, 1447
parallel processing model 1326
programming
mixed language 306
programming practices 430
program name 3203
programs 95, 101, 168, 257, 258, 261, 264, 356, 553, 573, 577, 748, 1479, 1480, 1897, 3404
advantages of internal procedures 261
advantages of modules 258
choosing development environment 95
debugging multithread 168
Fortran executables 257
mixed-language issues in 264
multithread 101, 1479
option linking as DLL 553
option maximizing speed in 577
option specifying aliasing should be assumed in 573
option specifying non-Fortran 748
running within another program 3404
structuring 257
values returned at termination of 356
writing 1480
program structure 1745
program termination
values returned 356
program unit call procedures
table of 2249
program unit definition procedures
table of 2249
program units 1745, 1746, 1897, 1898, 1914, 2171, 2181, 2417, 2603, 2705, 3047, 3203, 3394, 3586, 3645
allowing access to module 3645
block data 2417
external subprograms 1914
function 2705
main 1898, 3203
module 3047
order of statements in 1746
returning control to 3394
scope of 2171
statement terminating 2603
subroutine 3586
program units *(continued)*
types of association for 2181
program units and procedures 1897
projects
errors during build 331
prompt
subroutine controlling for critical errors 3444
PROTECTED 3205
prototyping procedures 279
PSECT 3207
pseudorandom number generators
RAN 3352
RANDOM 3353, 3355, 3420
RANDOM_NUMBER 3357
RANDU 3362
subroutine changing seed for 3360, 3420
subroutine querying seed for 3360
PUBLIC 3208
public entities 3208, 3645
renaming 3645
PURE 2705, 3212, 3586
in functions 2705
in subroutines 3586
pure procedures 2596, 2682, 2923, 3212
in FORALLs 2682
in interface blocks 2923
restricted form of 2596
PUTC 3215, 3293
POSIX version of 3293
PUTIMAGE 3216
PUTIMAGE_W 3216
PXF(type)GET 3220
PXF(type)SET 3221
PXFA(type)SET 3224
PXFACCESS 3226
PXFACHARGET 3223
PXFACHARSET 3224
PXFADBLGET 3223
PXFADBLSET 3224
PXFAGET 3223
PXFAINT8GET 3223
PXFAINT8SET 3224
PXFAINTGET 3223
PXFAINTSET 3224
PXFALARM 3227
PXFALGCLGET 3223
PXFALGCLSET 3224
PXFAREALGET 3223
PXFAREALSET 3224

PXFASTRGET 3223
PXFASTRSET 3224
PXFCALLSUBHANDLE 3228
PXFCFGETISPEED 3229
PXFCFGETOSPEED 3229
PXFCFSETISPEED 3230
PXFCFSETOSPEED 3231
PXFCHARGET 3220
PXFCHARSET 3221
PXFCHDIR 3231
PXFCHMOD 3232
PXFCHOWN 3233
PXFCLEARENV 3233
PXFCLOSE 3234
PXFCLOSEDIR 3234
PXFCONST 3235
PXFCREAT 3236
PXFCTERMID 3237
PXFDBLGET 3220
PXFDBLSET 3221
PXFDUP 3237
PXFDUP2 3237
PXFE(type)GET 3238
PXFE(type)SET 3239
PXFECHARGET 3238
PXFECHARSET 3239
PXFEDBLGET 3238
PXFEDBLSET 3239
PXFEINT8GET 3238
PXFEINT8SET 3239
PXFEINTGET 3238
PXFEINTSET 3239
PXFELGCLGET 3238
PXFELGCLSET 3239
PXFEREALGET 3238
PXFEREALSET 3239
PXFESTRGET 3238
PXFESTRSET 3239
PXFEXECV 3241
PXFEXECVE 3242
PXFEXECVP 3243
PXFEXIT 3244
PXFFASTEXIT 3244
PXFFCNTL 3245
PXFFDOPEN 3248
PXFFFLUSH 3250
PXFFGETC 3250
PXFFILENO 3251
PXFFORK 3252
PXFFPATHCONF 3254
PXFFPUTC 3256
PXFFSEEK 3257
PXFFSTAT 3258
PXFFTELL 3258
PXFGETARG 3259
PXFGETATTY 3260
PXFGETC 3260
PXFGETCWD 3261
PXFGETEGID 3261
PXFGETENV 3262
PXFGETEUID 3263
PXFGETGID 3263
PXFGETGRGID 3264
PXFGETGRNAM 3264
PXFGETGROUPS 3265
PXFGETLOGIN 3268
PXFGETPGRP 3269
PXFGETPID 3269
PXFGETPPID 3271
PXFGETPWNAM 3272
PXFGETPWUID 3273
PXFGETSUBHANDLE 3274
PXFGETUID 3275
PXFINT8GET 3220
PXFINT8SET 3221
PXFINTGET 3220
PXFINTSET 3221
PXFISBLK 3275
PXFISCHR 3276
PXFISCONST 3276
PXFISDIR 3277
PXFISFIFO 3278
PXFISREG 3278
PXFKILL 3279
PXFLGCLGET 3220
PXFLGCLSET 3221
PXFLINK 3280
PXFLOCALTIME 3281
PXFLSEEK 3282
PXFMKDIR 3283
PXFMKFIFO 3284
PXFOPEN 3284
PXFOPENDIR 3288
PXFPATHCONF 3289
PXFPAUSE 3291
PXFPIPE 3292
PXFPOSIXIO 3292
PXFPUTC 3293

PXFREAD 3294
PXFREADDIR 3295
PXFREALGET 3220
PXFREALSET 3221
PXFRENAME 3295
PXFREWINDDIR 3296
PXFRMDIR 3297
PXFSETENV 3297
PXFSETGID 3299
PXFSETPGID 3300
PXFSETSID 3301
PXFSETUID 3301
PXFSIGACTION 3302
PXFSIGADDSET 3303
PXFSIGDELSET 3304
PXFSIGEMPTYSET 3305
PXFSIGFILLSET 3306
PXFSIGISMEMBER 3306
PXFSIGPENDING 3307
PXFSIGPROCMASK 3308
PXFSIGSUSPEND 3309
PXFSLEEP 3310
PXFSTAT 3310
PXFSTRGET 3220
PXFSTRSET 3221
PXFSTRUCTCOPY 3311
PXFSTRUCTCREATE 3312
PXFSTRUCTFREE 3318
PXFSYSCONF 3319
PXFTCDRAIN 3322
PXFTCFLOW 3322
PXFTCFLUSH 3323
PXFTCGETATTR 3324
PXFTCGETPGRP 3325
PXFTCSENDBREAK 3326
PXFTCSETATTR 3326
PXFTCSETPGRP 3327
PXFTIME 3328
PXFTIMES 3329
PXFTTYNAM 3333
PXFUCOMPARE 3333
PXFUMASK 3334
PXFUNAME 3334
PXFUNLINK 3335
PXFUTIME 3335
PXFWAIT 3336
PXFWAITPID 3338
PXFWIFEXITED 3340
PXFWIFSIGNALED 3342
PXFWIFSTOPPED 3342
PXFWRITE 3343

Q

Q 2080
    edit descriptor 2080
QABS 2321
QACOS 2326
QACOSD 2327
QACOSH 2328
QARCOS 2326
QASIN 2350
QASIND 2351
QASINH 2352
QATAN 2365
QATAN2 2365
QATAN2D 2367
QATAND 2368
QATANH 2368
QCMPLX 3344
QCONJG 2482
QCOS 2486
QCOSD 2487
QCOSH 2488
QCOTAN 2488
QCOTAND 2489
QDIM 2539
QERF 2640
QERFC 2641
QEXP 2647
QEXT 3345
QEXTD 3345
QFLOAT 3346
QIMAG 2330
QINT 2331
QLOG 2979
QLOG10 2980
QMAX1 2995
QMIN1 3028
QMOD 3040
QNINT 2342
QNUM 3347
QRANSET 3347
QREAL 3348
QSIGN 3509
QSIN 3511
QSIND 3512

QSINH 3513
QSORT 3348
QSQRT 3561
QTAN 3598
QTAND 3599
QTANH 3599
Qtrapuv compiler option 359
quad-precision product
  function producing 2589
quick reference 1258, 1302, 1500, 1520
  automatic optimizations 1302
  compiler reports 1258
  IPO options 1500
  PGO options 1520
quick reference summary
  of Linux options 1178
  of Mac OS X options 1178
  of Windows options 1127
quick sort
  subroutine performing on arrays 3348
QuickWin
  initializing with user-defined settings 2900
QuickWin functions
  ABOUTBOXQQ 2320
  APPENDMENUQQ 2345
  CLICKMENUQQ 2455
  DELETEMENUQQ 2527
  FOCUSQQ 2667
  GETACTIVEQQ 2714
  GETEXITQQ 2753
  GETHWNDQQ 2767
  GETUNITQQ 2797
  GETWINDOWCONFIG 2799
  GETWSIZEQQ 2806
  INCHARQQ 2892
  INITIALIZEFONTS 2899
  INITIALSETTINGS 2900
  INQFOCUSQQ 2902
  INSERTMENUQQ 2907
  MESSAGEBOXQQ 3026
  MODIFYMENUFLAGSQQ 3042
  MODIFYMENUROUTINEQQ 3043
  MODIFYMENUSTRINGQQ 3045
  PASSDIRKEYSQQ 3150
  REGISTERMOUSEEVENT 3383
  RGBTOINTEGER 3399
  SETACTIVEQQ 3427
  SETEXITQQ 3445
  SETMOUSECURSOR 3466
QuickWin functions *(continued)*
  SETWINDOWCONFIG 3491
  SETWINDOWMENUQQ 3497
  SETWSIZEQQ 3502
  UNREGISTERMOUSEEVENT 3641
  WAITONMOUSEEVENT 3664
QuickWin procedures
  table of 2279
QuickWin routines
  predefined 2345, 2907, 3043
QuickWin subroutines
  INTEGERTORGB 2917
  SETMESSAGEQQ 3464
quotation mark editing 2083

R

RADIX 2224, 2225, 3350
  function returning 3350
  in integer model 2224
  in real model 2225
RAISEQQ 3351
RAN 3352
RAND 3353
RANDOM 3353, 3355
RANDOM_NUMBER 3357, 3360
  subroutine modifying or querying the seed of 3360
RANDOM_SEED 3360
random access I/O 208
random number generators
  IRAND 3562
  RAND 3562
  subroutine seeding 3562
random number procedures
  table of 2262
random numbers
  DRAND 2590
  DRANDM 2590
  function returning double-precision 2590
  IRAND 2936
  IRANDM 2936
  RAN 3352
  RAND and RANDOM 3353
  RANDOM 3355
  RANDOM_NUMBER 3357
  RANDU 3362
RANDU 3362
RANF 3363

RANGE 3363
ranges
  for complex constants 171
  for integer constants 171
  for logical constants 171
  for real constants 171
RANGET 3364
RANSET 3364
READ 1630, 1664, 1671, 2116, 3365
  and alignment 1671
  efficient use of 1630
  specifier for INQUIRE 2116
  using for little-to-big endian conversion 1664
READONLY 2140
READWRITE 2117
REAL 140, 1769, 2049, 2513, 3368, 3370, 3371
  compiler directive 140, 3370
  data type 1769, 3368
  editing 2049
  function 3371
  function converting to double precision 2513
REAL(16) 1729, 1769, 1773
  constants 1773
  representation 1729
REAL(4) 1728, 1769, 1771
  constants 1771
  representation 1728
REAL(8) 1729, 1769, 1772
  constants 1772
  representation 1729
REAL*16 1769
REAL*4 1769
REAL*8 1769
real and complex editing 2049
real constants
  rules for 1770
real conversion
  function performing 3371
real-coordinate graphics
  function converting to double precision 2513
  function converting to quad precision 3345
real data
  directive specifying default kind 3370
  function returning kind type parameter for 3423
  model for 2225
real data type 171, 1728, 1729, 1769, 1770, 1771, 1772, 1773, 2186, 2225, 2513, 3368
  constants 1770, 1771, 1772, 1773
  default kind 1769
real data type *(continued)*
  function converting to double precision 2513
  models for 2225
  native IEEE* representation 1728
  range for REAL*4 1728
  range for REAL*8 1729
  ranges for 171
  storage 2186
real editing 2049, 2050, 2052, 2054, 2056, 2058
  conversion 2058
  E and D 2052
  EN 2054
  engineering notation 2054
  ES 2056
  F 2050
  G 2058
  scientific notation 2056
  with exponents 2052
  without exponents 2050
real model 2225, 2635, 2649, 2693, 2820, 3612
  function returning exponent part in 2649
  function returning fractional part in 2693
  function returning largest number in 2820
  function returning number closest to unity in 2635
  function returning smallest number in 3612
real numbers
  directive specifying default kind 3370
  function resulting in single-precision type 3371
  function returning absolute spacing of 3527
  function returning ceiling of 2441
  function returning class of IEEE 2691
  function returning difference between 2539
  function returning floor of 2663
  function returning fractional part for model of 3448
  function returning scale of model for 3409
  function rounding 2342
  function truncating 2331
real-time clock
  subroutine returning data from 3595
real values
  transferring 2049, 2050, 2058
  transferring in exponential form 2052
  transferring using engineering notation 2054
  transferring using scientific notation 2056
REC 1985, 3365, 3673
  specifier for READ 3365
  specifier for WRITE 3673
reciprocal
  function returning 3402

RECL 1630, 2117, 2141
    specifier for INQUIRE 2117
    specifier for OPEN 1630, 2141
RECORD 3373
record access 223
record I/O 238
record I/O statement specifiers 235
record length 222
record number
    identifying for data transfer 1985
record position
    specifying 237
records
    function checking for end-of-file 2629
    option specifying padding for 807, 972
    repositioning to first 3397
    rewriting 3398
    specifying line terminator for formatted files 213
    statement to delete 2526
    statement writing end-of-file 2605
    types of 213, 1979
RECORDSIZE 2143
record specifier 1985, 2200
    alternative syntax for 2200
record structure fields 2204, 2609, 3579
    references to 2204
record structure items
    directive specifying starting address of 3136
record structures 305, 2201, 2209, 2609, 2623, 3124, 3373, 3579, 3634
    aggregate assignment 2209
    converting to Fortran 95/90 derived types 2201
    directive modifying alignment of data in 3124
    in mixed-language programming 305
    MAP declarations in 2623, 3634
    UNION declarations in 2623, 3634
record transfer 225
RECORDTYPE 2117, 2143
    specifier for INQUIRE 2117
    specifier for OPEN 2143
record type
    converting nonnative data using OPEN defaults 187
record types 213
RECTANGLE 3374
RECTANGLE_W 3374
rectangles
    functions drawing 3374
    subroutines storing screen image defined by 2768
recursion 3377
RECURSIVE 2705, 3377, 3586
    in functions 2705
    in subroutines 3586
recursive execution
    option specifying 1047
recursive procedures 2705, 3377, 3586
    as functions 2705
    as subroutines 3586
redirecting output 113
redistributable libraries 315
REDUCTION 1342, 1348, 1360, 1371, 1392, 2579, 3139, 3145, 3146, 3378, 3418
    in DO directive 2579
    in PARALLEL directive 3139
    in PARALLEL DO directive 3145
    in PARALLEL SECTIONS directive 3146
    in SECTIONS directive 3418
    in the DO directive 1371
    summary of data scope attribute clauses 1342, 1392
    used in PARALLEL directive 1360
    using 1348
    variables 1348
reductions in loops 1584
reentrancy protection
    function controlling 2681
REFERENCE 2389, 2393
    option for ATTRIBUTES directive 2389, 2393
references
    function 1914
    module 1899
    to elemental intrinsic procedures 1934
    to generic intrinsic functions 1930
    to generic procedures 1930
    to nonestablished names 2180
    to non-Fortran procedures 1935
register allocation 1606
register allocator
    option selecting method for partitioning 790, 959
REGISTERMOUSEEVENT 3383
relational expressions 1823
relational operators 1823
relative errors 1727
relative files 2526, 3702
    statement to delete records from 2526
relative spacing
    function returning reciprocal of 3402
remainder
    functions returning 3040

REMAPALLPALETTERGB 3384
REMAPPALETTERGB 3384
Remapping RGB values for video hardware 3384
removed compiler options 457
RENAME 3387
RENAMEFILEQQ 3388
REPEAT 3390
repeatable edit descriptors 2031, 2039
repeat specification 2039, 2068, 2083, 2086
    nested and group 2086
replicated arrays
    function creating 3559
report generation
    dynamic profile counters 1579
    improving 1636
    Intel extension 1402
    OpenMP* run-time 1398
    profile information 1579
    slowing down 1658
    timing 1398
    using compiler commands 1260
    using xi* tools 1510
report software pipelining (SWP) 1288
RESHAPE 3390
resolving generic references 2176
resolving procedure references 2175
resolving specific references 2179
response files 157, 158
    using 158
restricted expressions 1831
restricting optimization 1314
restrictions
    in using traceback information 413
RESULT 2705, 3377, 3392
    defining explicit interface 3377
    keyword in functions 2705
result name 2705, 3392
    in functions 2705
result variables 1937, 2627, 2705, 3392
    in ENTRY 2627
    requiring explicit interface 1937
ResumeThread 1491
RETURN 3394, 3405
    retaining data after execution of 3405
return values
    placement in argument list 290, 301
REWIND 3397
REWRITE 3398
RGB color
    subroutine converting into components 2917
RGB color values
    function converting integer to 3399
    function remapping 3384
    function returning current 2727
    function returning for multiple pixels 2782
    function returning for pixel 2779
    function returning text 2790
    function setting current 3436
    function setting for multiple pixels 3474
    function setting for pixel 3470
    function setting text 3478
RGB components
    subroutine converting color into 2917
RGBTOINTEGER 3399
right shift
    function performing arithmetic 2941
    function performing circular 2942
    function performing logical 2947
RINDEX 3401
RNUM 3402
root procedures
    table of 2269
rounding
    function performing 3069
rounding errors 1727, 1733
    machine epsilon 1727
    magnitude of 1727
    relative 1727
    ULPs 1727
rounding flags 1711
routine entry
    option specifying the stack alignment to use on 575
routine entry and exit points
    option determining instrumentation of 586, 912
routines
    module 2229
    OpenMP Fortran 2230
    run-time library 2229
    storing in shareable libraries 261
RRSPACING 3402
RSHFT 2943
RSHIFT 2943
RTC 3403
running applications
    from the command line 118
RUNQQ 3404

run-time checking 1310
run-time environment
    function cleaning up 2673
    function initializing 2674
run-time environment variables 132
run-time error messages 353, 359, 361, 412
    format 353
    locating 359
    locating cause 359
    using traceback information 412
    where displayed 353
run-time error processing
    default 353
run-time errors
    functions returning most recent 2769, 2770
Run-Time Library (RTL)
    error processing performed by 353
    function controlling reentrancy protection for 2681
    option searching for unresolved references in multithreaded 726, 739, 1077
    option searching for unresolved references in single-threaded 733
    option specifying which to link to 712
    requesting traceback 412
run-time performance
    improving 1693
run-time routines
    COMMITQQ 2471
    FOR_DESCRIPTOR_ASSIGN 2668
    FOR_GET_FPE 2672
    for_rtl_finish_ 2673
    for_rtl_init_ 2674
    FOR_SET_FPE 2674
    FOR_SET_REENTRANCY 2681
    GERROR 2712
    GETCHARQQ 2723
    GETEXCEPTIONPTRSQQ 2751
    GETSTRQQ 2787
    PEEKCHARQQ 3158
    PERROR 3159
    TRACEBACKQQ 3612

S

S 2072
    edit descriptor 2072
S_floating format 1771
sample code
    array_calc program 110
sample of timing 1643
sample programs
    and traceback information 415
SAVE 3405
SAVEIMAGE 3408
SAVEIMAGE_W 3408
SAVE value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
scalar clean-up iterations 1475, 1660
    allocation of 1660
scalar replacement
    option enabling during loop transformation 1015, 1056
    option using aggressive multi-versioning check for 783, 952
scalars 1798, 1799, 1800, 3504, 3704
    as subobjects 1798
    as variables 1798
    function returning shape of 3504
    typing of 1799, 1800
scalar variables
    data types of 1799
    option allocating to the run-time stack 496, 848
SCALE 3409
scale factor 2074
scale factor editing (P) 2074
SCAN 3410
SCANENV 3412
SCHEDULE 1342, 1371, 2579, 3145
    AUTO 2579
    DYNAMIC 1342
    GUIDED 1342
    in DO directive 2579
    in PARALLEL DO directive 3145
    RUNTIME 1342
    STATIC 1342
    using in DO directives 1371
    using to specify types and chunk sizes 1342
scientific-notation editing (ES) 2056
scope 2171, 2175
    of unambiguous procedure references 2175
scoping units 1746, 2171, 3645
    statements restricted in 1746
    with more than one USE 3645
scratch files 209
screen area
    erasing and filling 2451

screen images
subroutines storing rectangle 2768
screen output
displaying 3194
SCROLLTEXTWINDOW 3412
SCWRQQ 3415
SECNDS 3416, 3417
seconds
function returning since Greenwich mean time 3403
function returning since midnight 3416
function returning since TIMEF was called 3611
SECTION 3418
SECTIONS 3418
SEED 3420
seeds
subroutine changing for RAND and IRAND 3562
subroutine changing for RANDOM 3420
subroutine modifying or querying for RANDOM_NUMBER 3360
subroutine returning 2938, 3364
subroutine setting 2592, 2938, 3364
SELECT CASE 2433
SELECTED_CHAR_KIND 3422
SELECTED_INT_KIND 3422
SELECTED_REAL_KIND 3423
semaphores 1488
semicolon (;)
as source statement separator 1752
SEQUENCE 3425
SEQUENTIAL 2118
specifier for INQUIRE 2118
sequential access mode 1979
sequential file access 210
sequential files 1979, 2407
positioning at beginning 2407
sequential READ statements 1998, 1999, 2000, 2003, 2011
rules for formatted 1999
rules for list-directed 2000
rules for namelist 2003
rules for unformatted 2011
sequential WRITE statements 2017, 2018, 2019, 2021, 2025
rules for formatted 2018
rules for list-directed 2019
rules for namelist 2021
rules for unformatted 2025
serial execution 1352
serial port I/O routines
SPORT_CANCEL_IO 3530
SPORT_CONNECT 3531
SPORT_CONNECT_EX 3533
SPORT_GET_HANDLE 3535
SPORT_GET_STATE 3536
SPORT_GET_STATE_EX 3537
SPORT_GET_TIMEOUTS 3540
SPORT_PEEK_DATA 3542
SPORT_PEEK_LINE 3543
SPORT_PURGE 3544
SPORT_READ_DATA 3545
SPORT_READ_LINE 3546
SPORT_RELEASE 3548
SPORT_SET_STATE 3549
SPORT_SET_STATE_EX 3550
SPORT_SET_TIMEOUTS 3553
SPORT_SHOW_STATE 3554
SPORT_SPECIAL_FUNC 3556
SPORT_WRITE_DATA 3557
SPORT_WRITE_LINE 3558
SET_EXPONENT 3448
SETACTIVEQQ 3427
SETBKCOLOR 3428
SETBKCOLORRGB 3429
SETCLIPRGN 3431
SETCOLOR 3434
SETCOLORRGB 3436
SETCONTROLFPQQ 1710, 1711, 3438
example of 1711
using to set the control word value 1710
SETDAT 3441
SETENVQQ 3442
SETERRORMODEQQ 3444
SETEXITQQ 3445
SETFILEACCESSQQ 3448
SETFILETIMEQQ 3450
SETFILLMASK 3451
SETFONT 3455
SETGTEXTROTATION 3460
SETLINESTYLE 3462
SETMESSAGEQQ 3464
SETMOUSECURSOR 3466
SETPIXEL 3469
SETPIXEL_W 3469
SETPIXELRGB 3470
SETPIXELRGB_W 3470
SETPIXELS 3473
SETPIXELSRGB 3474

SETTEXTCOLOR 3477
SETTEXTCOLORRGB 3478
SETTEXTCURSOR 3480
SETTEXTPOSITION 3483
SETTEXTWINDOW 3484
SetThreadPriority 1481
SETTIM 3485
setting
    compiler options on the command line 137
    environment variables 129
SETVIEWORG 3487
SETVIEWPORT 3488
SETWINDOW 3489
SETWINDOWCONFIG 3491
SETWINDOWMENUQQ 3497
SETWRITEMODE 3498
SETWSIZEQQ 3502
SHAPE 3504
shape of array
    function constructing new 3390
    function returning 3504
    statement defining 2540
shapes
    subroutine returning pattern used to fill 2759
SHARE 2119, 2144
    specifier for INQUIRE 2119
    specifier for OPEN 2144
shareable libraries 261
SHARED 2146, 2521, 3139, 3145, 3146, 3507
    clause in PARALLEL directive 3139
    clause in PARALLEL DO directive 3145
    clause in PARALLEL SECTIONS directive 3146
    specification in DEFAULT clause 2521
    specifier for OPEN 2146
shared libraries 321
shared memory access
    requesting threaded program execution 101
shared object
    option producing a dynamic 1057
shared scalars 1379
shared scoping 1352
shared variables 1348
sharing
    specifying file 2144
shell
    function sending system command to 3593
SHIFTL 3507
SHIFTR 3508
SHORT 3508
short field termination 2066
side effects of procedures
    preventing 3212
SIGN 3509
signal 410
SIGNAL 3513
signal handling 410
SIGNALQQ 3516
signals
    debugging 164
    function changing the action for 3513
    function sending to executing program 3351
    function sending to process ID 2954
signed infinity 1722
sign editing 2071, 2072
    S 2072
    SP 2072
    SS 2072
signed zero 1722
significant digits
    function returning number of 2538
SIN 3511
SIND 3512
sine
    function returning 3511, 3512
    function returning hyperbolic 3513
    function with argument in degrees 3512
    function with argument in radians 3511
SINGLE 3520
single-precision constants
    option evaluating as double precision 616
single-precision real 1611, 1636, 1769, 2910
    function converting to truncated integer 2910
SINH 3513
SIZE 1989, 3365, 3521
    specifier for READ 3365
SIZEOF 3522
size of arrays
    function returning 3521
    system parameters for 438
size of executable programs
    system parameters for 438
slash editing 2077
SLEEP 3523
SLEEPQQ 3524
SMP systems 1447
SNGL 3371
SNGLQ 3371

software pipelining 1242, 1288, 1591, 1605
affect of LOOP COUNT on 1591
for IA-64 architecture based applications 1605
optimization 1605
reports 1288
sorting a one-dimensional array 3525
SORTQQ 3525
source code 276, 429, 1636, 1754, 1757, 1761
case-sensitivity of names 276
fixed and tab form of 1757
free form of 1754
porting between systems 429
useable for all source forms 1761
source code analysis 336, 342, 344, 345, 346, 347
C/C++-specific analysis 346
Fortran-specific analysis 345
interprocedural analysis 342
local program analysis 344
OpenMP* analysis 347
source code format 1752
source code useable for all source forms 1761
source comments 1752
source files
compiling and linking a single 107
source forms
combining 1761
fixed and tab 1757
free 1754
overview of 1745, 1752
source lines
padding fixed and tab source 1757
padding free source 1754
SP 2072
edit descriptor 2072
space
allocating for arrays and pointer targets 2338
deallocating for arrays and pointer targets 2517
disassociating for pointers 3112
SPACING 3527
speaker
subroutine sounding 2410
speaker procedures
table of 2264
specialized code 1242, 1310, 1611
specification expressions 1831
inquiry functions allowed in 1831
transformational functions allowed in 1831
specifications
table of procedures for data 2250
specification statements 1845
specific names
references to 2179
specific references 2179
specifying carriage control 2129
specifying file numeric format 2130
precedence 2130
specifying file position 2139
specifying file sharing 2144
specifying file structure 2136
specifying symbol visibility 1669
specifying variables 1799, 2250
table of procedures 2250
SPLITPATHQQ 3528
SPORT_CANCEL_IO 3530
SPORT_CONNECT 3531
SPORT_CONNECT_EX 3533
SPORT_GET_HANDLE 3535
SPORT_GET_STATE 3536
SPORT_GET_STATE_EX 3537
SPORT_GET_TIMEOUTS 3540
SPORT_PEEK_DATA 3542
SPORT_PEEK_LINE 3543
SPORT_PURGE 3544
SPORT_READ_DATA 3545
SPORT_READ_LINE 3546
SPORT_RELEASE 3548
SPORT_SET_STATE 3549
SPORT_SET_STATE_EX 3550
SPORT_SET_TIMEOUTS 3553
SPORT_SHOW_STATE 3554
SPORT_SPECIAL_FUNC 3556
SPORT_WRITE_DATA 3557
SPORT_WRITE_LINE 3558
SPREAD 3559
SQRT 3561
square root
function returning 3561
SRAND 3562
SS 2072
edit descriptor 2072
SSE 1459, 1611
optimizing 1611
SSE2 1459
SSWRQQ 3564
stack
option disabling checking for routines in 660
option enabling probing 853
option specifying reserve amount 569

stack *(continued)*
size for threads 1481
stack alignment
option specifying for functions 1016
stack probing
option enabling 853
stacks 1658, 1660
stack storage
allocating variables to 2402
stack variables
option initializing to NaN 642, 1024
standard directories
option removing from include search path 1116
standard error
redirecting command-line output 113
standard error output file 331
standard error stream
subroutine sending a message to 3159
standard output
redirecting command-line output 113
standards
Fortran 95 or Fortran 90 checking 121, 430
language 429
STAT 3564
statement field
option specifying the length of 565
statement functions 1636, 1897, 2191, 3569
statement labels 1752
statements
ACCEPT 2323
ALLOCATABLE 2337
ALLOCATE 2338
arithmetic IF 2873
ASSIGN 2352
assigned GO TO 2810
assignment 1833
ASYNCHRONOUS 2363
AUTOMATIC 2402
BACKSPACE 2407
BIND 2414
BLOCK DATA 2417
BYTE 2424
CALL 2429
CASE 2433
CHARACTER 2445
classes of 1746
CLOSE 2457
COMMON 2473
COMPLEX 2478
statements *(continued)*
computed GO TO 2811
conditional execution based on logical expression 2875
conditionally executing groups of 2876
CONTAINS 2483
CONTINUE 2484
control 1883
CYCLE 2498
DATA 2500
data transfer 1979
DEALLOCATE 2517
declaration 1845
DECODE 2519
DEFINE FILE 2523
DELETE 2526
derived type 2530, 2617, 3620
DIMENSION 2540
DO 2575
DOUBLE COMPLEX 2587
DOUBLE PRECISION 2588
DO WHILE 2584
ELSE WHERE 2600, 3666
ENCODE 2601
END 2603
END DO 2604
ENDFILE 2605
END WHERE 2626
ENTRY 2627
EQUIVALENCE 1863, 2636
executable 1746
EXIT 2645
EXTERNAL 2650
FIND 2655
FLUSH 2666
FORALL 2682
FORMAT 2685
FUNCTION 2705
IF - arithmetic 2873
IF construct 2876
IF - logical 2875
IMPLICIT 2889
IMPORT 2891
input/output 2095
INQUIRE 2903
INTEGER 2915
INTENT 2919
INTERFACE 2923
INTERFACE TO 2926

statements *(continued)*
INTRINSIC 2927
LOGICAL 2982
MAP 2623, 3634
MODULE 3047
MODULE PROCEDURE 3053
NAMELIST 3064
nonexecutable 1746
NULLIFY 3112
OPEN 3115
OPTIONAL 3118
OPTIONS 3122
order in program units 1746
PARAMETER 3148
PAUSE 3156
POINTER 3163
POINTER - Integer 3166
PRINT 3194
PRIVATE 3196
PROGRAM 3203
PUBLIC 3208
READ 3365
REAL 3368
RECORD 3373
repeatedly executing 2575
repeatedly executing while true 2584
restricted in scoping units 1746
RETURN 3394
REWIND 3397
REWRITE 3398
SAVE 3405
SELECT CASE 2433
SEQUENCE 3425
specification 1845
statement function 3569
STATIC 3572
STOP 3575
STRUCTURE 2609, 3579
SUBROUTINE 3586
TARGET 3600
TYPE 2530, 2617, 3620
type declaration 1846, 3626
unconditional GO TO 2813
UNION 2623, 3634
USE 3645
VALUE 3653
VIRTUAL 3661
VOLATILE 3661
WAIT 3663
statements *(continued)*
WHERE 3666
WRITE 3673
statement scope 2171
statement separator 1752
state messages
subroutine setting 3464
STATIC 3572
static libraries 261, 319, 1066
option invoking tool to generate 1066
static storage
allocating variables to 3572
STATUS 2146
specifier for OPEN 2146
status messages
subroutine setting 3464
status of graphics routines
function returning 2814
STATUS specifier for CLOSE 2457
status word
setting and retrieving floating-point 1706
subroutine clearing exception flags in floating-point 2452
subroutines returning floating-point 2785, 3564
STDCALL 2378, 2390
option for ATTRIBUTES directive 2378, 2390
STOP 3575
storage
association 2186, 2636
defining blocks of 2473
dynamically allocating 2338
freeing 2517
function returning byte-size of 3522
sequence 2186
sharing areas of 2636
units 2186
storage association 2186
using ENTRY 2186
storage item
function returning address of 2977
storage sequence 2186
storage units 2186
storing data per thread 1491
strategies for optimization 1251
Stream_CR records 213
Stream_LF records 213, 1630
Streaming SIMD Extensions 1461, 1611, 1671
record 1671

streaming stores
option generating for optimization 797, 966
stream record type 213
STRICT 140, 3103, 3577
equivalent compiler option for 140
stride 1809
string edit descriptors 2083, 2084
apostrophe 2083
H 2084
quotation mark 2083
strings
function concatenating copies of 3390
function locating last nonblank character in 2975
function returning length minus trailing blanks 2962
function returning length of 2961
mixed-language programming 301
writing unknown length to file or device 2084
STRUCTURE 2609, 3579
structure components 1786
structure constructors 1790
structure declarations 2203
structures
derived-type 2530, 2617, 3620
record 2201, 2203
structuring your program 257
SUBMIT/DELETE value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
SUBMIT value for CLOSE(DISPOSE) or CLOSE(STATUS) 2457
subnormal numbers 1688
subobjects 1798
subprograms 263, 1914, 2417, 2603, 2650, 2705, 2927, 3047, 3053, 3394, 3586
BLOCK DATA 2417
effect of RETURN in 3394
function 2705
module 3047, 3053
statement returning control from 2603
subroutine 3586
user-written 1914
using as actual arguments 2650, 2927
SUBROUTINE 3586
subroutine references 2429
subroutines 1897, 1915, 1945, 1947, 2333, 2429, 2627, 2650, 3377, 3586
effect of ENTRY in 1945
ELEMENTAL keyword in 3586
EXTERNAL 2650
function running at specified time 2333
subroutines *(continued)*
general rules for 1915
intrinsic 1947
invoking 2429
PURE keyword in 3586
RECURSIVE keyword in 3377, 3586
statement specifying entry point for 2627
transferring control to 2429
subroutines in the OpenMP* run-time library 1398, 1406, 1447, 1591, 1595, 1596, 1602
for loop unrolling 1595
for OpenMP* 1406
for optimization 1591
for prefetching 1602
for vectorization 1596
parallel run-time 1447
subscript list 1800, 1804, 1807
referencing array elements 1804, 1807
subscript progression 1804
subscripts 1804
subscript triplets 1807, 1809
substrings 1783, 1865, 2898, 3401
function locating index of last occurrence of 3401
function returning starting position of 2898
making equivalent 1865
substructure declarations
for record structures 2204
SUM 3590
sum of array elements
function returning 3590
support
for symbolic debugging 165
SuspendThread 1491
suspension
of program execution 3156
SWP 1288, 1605, 3105, 3592
SWP reports 1288
using 1605
symbolic constants
defining floating-point status and control 1706
symbolic names 522, 1748
option associating with an optional value 522
symbol names
option using dollar sign when producing 1029
symbols
predefined preprocessor 153
symbol visibility 647, 1669
option specifying 647
specifying 1669

symbol visibility on Linux* 1669
symbol visibility on Mac OS* X 1669
synchronization 1242, 1326, 1360, 1364, 1402, 1447, 1479, 1481, 1491
  changing the number of 1360
  concepts for multithread applications 1479
  constructs 1364
  creating multithread applications 1479
  parallel processing model for 1326
  starting and stopping 1481
  synchronizing 1491
  thread-level parallelism 1242
  thread sleep time 1402
synchronizing multithread programs 1480, 1491
syntax
  for the ifort command 108
  option checking for correct 1070
SYSTEM 3593
SYSTEM_CLOCK 3595
system calls
  using to open files 2148
system codepage
  function returning number for 3081
system command
  function sending to command interpreter 3596
  function sending to shell 3593
system date
  function setting 3441
system errors
  subroutine returning information on 2642
system parameters for language elements 438
system procedures
  table of 2288
system prompt
  subroutine controlling for critical errors 3444
SYSTEMQQ 3596
system subprograms
  CPU_TIME 2492
  DATE 2506
  DATE_AND_TIME 2509
  EXIT 2646
  IDATE 2831
  SECNDS 3417
  SYSTEM_CLOCK 3595
  TIME 3608
system time
  function converting to ASCII string 2456, 2497
  intrinsic returning 3608
  subroutine returning 3609
system time *(continued)*
  subroutine setting 3485

T

T 2070
  edit descriptor 2070
T_floating format 1772
tab-format source lines 1760
tab source format 1757, 1760
  lines in 1760
TAN 3598
TAND 3599
tangent
  function returning 3598, 3599
  function returning hyperbolic 3599
  function with argument in degrees 3599
  function with argument in radians 3598
TANH 3599
TARGET 3600
targeting 1306, 1310, 1313
  IA-32 architecture processors 1306
  Intel(R) 64 architecture processors 1306
  Itanium(R) 2 processors 1313
  run-time checking 1310
targets
  allocation of pointer 1877
  assigning values to 1833, 2357
  associating with pointers 1840, 3600
  as variables 1840
  creating storage for 2338
  deallocation of pointer 1880
  declaration of 3600
  requiring explicit interface 1937
TASK 3602
task region
  directive defining 3602
TASKWAIT 3606
technical applications 1251
TEMP environment variable 129
temporary files 124
  option to keep 124
terminal
  subroutine specifying device name for 3620
terminal statements for DO constructs 2575
TerminateProcess 1495
TerminateThread 1481

terminating format control (
) 2079
terminating short fields of input data 2066
ternary raster operation constants 3216
test prioritization tool 1552
examples 1552
options 1552
requirements 1552
text
function controlling truncation of 3671
function controlling wrapping of 3671
function returning orientation of 2766
function returning width for use with OUTGTEXT 2764
subroutine sending to screen (including blanks) 3130, 3133
subroutine sending to screen (special fonts) 3130
text color
function returning RGB value of 2790
text color index
function returning 2789
function returning RGB value of 2790
function setting 3477
function setting RGB value of 3478
text cursor
function setting height and width of 3480
text files
line including 2895
text output
function returning background color index for 2718
function returning background RGB color for 2719
function setting background color index for 3428
function setting background RGB color for 3429
text position
subroutine returning 2792
subroutine setting 3483
text window
subroutine returning boundaries of 2793
subroutine scrolling the contents of 3412
subroutine setting boundaries of 3484
thread affinity 808, 975, 1418
option specifying 808, 975
threaded applications
option enabling analysis of 1019, 1072
threaded program execution
requesting 101
threading 1479
thread local storage 1491
thread pooling 1454
THREADPRIVATE 3607
thread routine format 1484
threads
compiling and linking multithread applications 101
thread stacks 1488
threshold control for auto-parallelization 1287, 1398, 1461
OpenMP* routines for 1398
reordering 1461
TIME 2333, 2509, 2795, 2808, 2948, 2987, 3022, 3079, 3485, 3608, 3609, 3640
ALARM function for subroutines 2333
function returning accounting of 3022
function returning for current locale 3079
routines returning current system 3608, 3609
subroutine returning 2509, 2795
subroutine returning Greenwich mean 2808
subroutine returning in array 2948
subroutine returning local zone 2987
subroutine setting system 3485
subroutine unpacking a packed 3640
time and date
routine returning as ASCII string 2652
subroutine packing values for 3138
subroutine returning 4-digit year 2509
subroutine returning current system 2509
TIMEF 3611
TINY 3612
TITLE 2147
specifier for OPEN 2147
TL 2071
edit descriptor 2071
TMPDIR environment variable 129
TMP environment variable 129
tool options 1532, 1552, 1561
code coverage tool 1532
profmerge 1561
proforder 1561
test prioritization 1552
tools 97, 99, 929, 969, 1532
default 97
option passing options to 969
option specifying directory for supporting 929
specifying alternative 99
topology maps 1418
total association 2186
TR 2071
edit descriptor 2071

traceback
function returning argument eptr for TRACEBACKQQ 2751
subroutine aiding in 3612
traceback compiler option 359
traceback information
obtaining with TRACEBACKQQ routine 426
option providing 1080
restrictions in using 413
sample programs 415
tradeoffs in using 413
using 412
TRACEBACKQQ 426, 3612
using 426
tradeoffs
in using traceback information 413
TRAILZ 3615
transcendental functions
option replacing calls to 578, 879
TRANSFER 3616
transfer of data
function performing binary 3616
transformational functions 1828, 1831, 1947, 2335, 2344, 2490, 2494, 2586, 2632, 2993, 2998, 3002, 3031, 3035, 3110, 3134, 3201, 3390, 3422, 3423, 3559, 3590, 3616, 3618, 3619, 3638
ALL 2335
allowed in initialization expressions 1828
allowed in specification expressions 1831
ANY 2344
COUNT 2490
CSHIFT 2494
DOT_PRODUCT 2586
EOSHIFT 2632
MATMUL 2993
MAXLOC 2998
MAXVAL 3002
MINLOC 3031
MINVAL 3035
NULL 3110
PACK 3134
PRODUCT 3201
REPEAT 3390
RESHAPE 3390
SELECTED_CHAR_KIND 3422
SELECTED_INT_KIND 3422
SELECTED_REAL_KIND 3423
SPREAD 3559
transformational functions *(continued)*
SUM 3590
TRANSFER 3616
TRANSPOSE 3618
TRIM 3619
UNPACK 3638
transportability of data 433
TRANSPOSE 3618
transposed arrays
function producing 3618
trigonometric functions 2269
trigonometric procedures 2269
TRIM 3619
troubleshooting 331, 435
during application development 435
TTYNAM 3620
twos complement
function returning length in 2888
TYPE 2148, 2530, 2617, 3194, 3620
for derived types 2530, 2617, 3620
specifier for OPEN 2148
type aliasability rules
option affecting adherence to 479, 847
type conversion procedures
table of 2267
type declarations 2203, 3626
within record structures 2203
type declaration statements 1846, 1847, 1849, 1852, 1853, 3626
array 1853
attributes in 3626
character 1849
derived 1852
double colon separator in 3626
initialization expressions in 3626
noncharacter 1847

U

UBC buffers 1630
UBOUND
in pointer assignment 1860
ULPs 1727
unaligned data 164, 1099, 1613
option warning about 1099
unambiguous generic procedure references 2175
unambiguous references 2175
unary operations 1818

unbuffered WRITEs 1630
uncalled routines
option warning about 1099
unconditional DO 2575
unconditional GO TO 2813
undeclared symbols
option warning about 1099
UNDEFINE 2522, 3632
undefined variables 1798
underflow 1611, 1660
underscore (_)
in names 1748
UNFORMATTED 2120
specifier for INQUIRE 2120
unformatted data
and nonnative numeric formats 181
unformatted direct-access READ statements 2013
unformatted direct-access WRITE statements 2027
unformatted direct files 208
unformatted files 187, 188, 190, 194, 195, 196, 208, 1630, 2136
converting nonnative data 187
direct-access 208
methods of specifying endian format 188
obtaining numeric specifying format 188
using /convert option to specify format 196
using environment variable method to specify format 190
using OPEN(CONVERT=) method to specify format 194
using OPTIONs/CONVERT to specify format 195
unformatted numeric data
option specifying format of 517
unformatted records 1979
unformatted sequential files 208
unformatted sequential READ statements 2011
unformatted sequential WRITE statements 2025
Unicode* characters 102
uninitialized variables
option checking for 511
UNION 2623, 3634
UNIT 197, 1983, 3365, 3673
specifier for READ 3365
specifier for WRITE 3673
using for external files 197
using for internal files 197
unit number
function testing whether it's a terminal 2939
unit number 6
function writing a character to 3215
unit numbers 197
units 1983, 2457, 2903, 3115
disconnecting 2457
opening 3115
statement requesting properties of 2903
UNIT specifier for CLOSE 2457
UNLINK 3637
UNPACK 3638
unpacked array
function creating 3638
UNPACKTIMEQQ 3640
UNREGISTERMOUSEEVENT 3641
UNROLL 3108, 3643
UNROLL_AND_JAM 3644
UNTIED clause 3602
unused variables
option warning about 1099
unvectorizable copy 1461
usage rules 1352
USE 3645
use association 2183
user
function returning group ID of 2764
function returning ID of 2796
subroutine returning login name of 2774
user-defined procedures
elemental 2596
keyword preventing side effects in 3212
pure 3212
user-defined types 305, 1784
mixed-language programming 305
user-defined TYPE statement 2530, 2617, 3620
user functions 1242, 1346, 1352, 1360, 1371, 1379, 1398, 1447, 1512, 1514, 1530, 1574, 1611, 1630, 1636, 1638, 1643, 1660
automatic 1660
auto-parallelization 1242, 1447
dynamic libraries 1398
EQUIVALENCE statements 1636
floating-point conversions 1611
formatted or unformatted files 1630
implied-DO loops 1630
intrinsics 1638
length of 1630
loop assigns for 1346
memory 1630
noniterative worksharing SECTIONS 1371

user functions *(continued)*
OpenMP* 1379
PGO environment 1574
private scoping for 1352
profile-guided optimization 1530
slow arithmetic operators 1636
timing for an application 1643
unbuffered WRITEs 1630
worksharing 1360
user ID
function returning 2796
USEROPEN 2148
USEROPEN specifier 238
user-written subprograms 1914
using an external user-written function to open files 2148
using the compiler and linker from the command line 107
using the IFPORT portability module 325
utilities 1532, 1561
profmerge 1561
proforder 1561

V

VALUE 2389, 2393, 3653
option for ATTRIBUTES directive 2389, 2393
variable format expressions 2086
variables 283, 589, 691, 746, 928, 934, 1012, 1042, 1046, 1053, 1121, 1748, 1750, 1763, 1779, 1798, 1799, 1800, 1817, 1833, 1840, 1846, 2250, 2352, 2357, 2371, 2402, 2473, 2500, 2518, 2522, 2530, 2617, 2705, 2889, 3064, 3392, 3405, 3572, 3620, 3632
allocating to stack storage 2402
allocating to static storage 3572
assigning initial values to 2500
assigning value of label to 2352
assigning values to 1833, 2357
associating with group name 3064
automatic 2402
character 1779
data types of scalar 1799
declaring automatic 2402
declaring derived-type 2530, 2617, 3620
declaring static 3572
declaring type for 1846
directive creating symbolic 2522, 3632
variables *(continued)*
directive declaring properties of 2371
directive generating warnings for undeclared 2518
directive testing value of 2522, 3632
direct sharing of 2473
explicit typing of 1799
giving initial values to 2500
how they become defined or undefined 1798
implicit typing of 1800
initializing 2500
length of name 1748
namelist 3064
on the stack 2402
option initializing to zero 1042, 1121
option placing in DATA section 746, 934
option placing in static memory 1012, 1053
option saving always 589, 928
option specifying default kind for integer 691
option specifying default kind for logical 691
option specifying default kind for real 1046
referencing 1817
result 2705, 3392
retaining in memory 3572
saving values of 3405
statement defining default types for user-defined 2889
static 3572
storage association of 2473
table of procedures that declare 2250
targets as 1840
truncation of values assigned to 1833
typing of scalar 1799, 1800
undefined 2518
using keyword names for 1750
using modules in mixed-language programming 283
VARYING 2394
option for ATTRIBUTES directive 2394
VAXD 2130
value for CONVERT specifier 2130
VAXG 2130
value for CONVERT specifier 2130
VECTOR ALIGNED 3654, 3659
VECTOR ALWAYS 3109, 3655
VECTOR ALWAYS directive 1596
vector copy 1242, 1294, 1459, 1461, 1475, 1596
examples 1475
options 1459
options for 1242

vector copy *(continued)*
overview 1459
programming guidelines 1459, 1461
reports 1294
support for 1596
VECTOR directive 1596
vectorization
option disabling 1032, 1090
option setting threshold for loops 1036, 1094
vectorizer
option controlling diagnostics reported by 1034, 1092
vectorizing 1461, 1466, 1519
loops 1466, 1519
VECTOR NONTEMPORAL 3657, 3658
VECTOR NONTEMPORAL directive 1596
vectors
function performing dot-product multiplication of 2586
subscripts in 1807, 1810
vector subscripts 1807, 1810
VECTOR TEMPORAL 3657, 3658
VECTOR UNALIGNED 3654, 3659
VERIFY 3660
version
option displaying for driver and compiler 1107
option displaying information about 716
option saving in executable or object file 1017, 1062
viewport area
subroutine erasing and filling 2451
subroutine redefining 3488
viewport-coordinate origin
subroutine moving 3487
subroutine setting 3488
viewport coordinates
functions filling (color index) 2658
functions filling (RGB) 2661
subroutine converting to physical coordinates 2775
subroutine converting to Windows coordinates 2804
subroutines converting from physical coordinates 2798
viewport origin
subroutine moving 3487
VIRTUAL 3661
VMS* Compatibility
option specifying 1095
VOLATILE 1630, 3661
using for loop collapsing 1630

W

WAIT 3663
WaitForMultipleObjects 1491
WaitForSingleObject 1491
WAITONMOUSEEVENT 3664
warn compiler option 331
warning messages
controlling issue of 331
directive generating for undeclared variables 2518
directive modifying for data alignment 3124
floating-point overflow (run-time) 1714
floating-point underflow (run-time) 1714
watch compiler option 331
WB compiler option 331
WHERE 2600, 2626, 3666
ELSE WHERE block in 2600
statement ending 2626
WHILE 2584
whole arrays 1803
whole program analysis 1497
WINABOUT 2345
predefined QuickWin routine 2345
WINARRANGE 2345
predefined QuickWin routine 2345
WINCASCADE 2345
predefined QuickWin routine 2345
WINCLEARPASTE 2345
predefined QuickWin routine 2345
WINCOPY 2345
predefined QuickWin routine 2345
window
function making child active 3427
function returning unit number of active child 2714
subroutine scrolling the contents of text 3412
window area
function defining coordinates for 3489
subroutine erasing and filling 2451
window handle
function returning unit number of 2797
Windows
function converting unit number to handle 2767
function returning position of 2806
function returning properties of 2799
function returning size of 2806
function returning unit number of 2797
function setting position of 3502
function setting properties of child 3491
function setting size of 3502

Windows *(continued)*
setting focus to 2667
subroutine returning boundaries of text 2793
subroutine scrolling the contents of text 3412
subroutine setting boundaries of text 3484
Windows* API
BitBlt 3216
CreateFile 2148, 3535
CreateFontIndirect 3455, 3491
CreateProcess 3593, 3596
EscapeCommFunction 3556
GetEnvironmentVariable 2748
GetExceptionInformation 3612
PurgeComm 3544
SetEnvironmentVariable 2748
SetFileApisToANSI 3096
SetFileApisToOEM 3096
SetROP2 3498
Windows* applications
option creating and linking 1108
Windows* bitmap file
function saving an image into 3408
Windows* compiler options
/? 663
/1 764, 935
/4I2 691
/4I4 691
/4I8 691
/4L72 565
/4L80 565
/4Na 498
/4Naltparam 478
/4Nb 511
/4Nd 1099
/4Nf 588
/4Nportlib 469, 471
/4Ns 1063
/4R16 1046
/4R8 1046
/4Ya 498
/4Yaltparam 478
/4Yb 511
/4Yd 1099
/4Yf 638
/4Yportlib 469, 471
/4Ys 1063
/align 472
/allow:fpp_comments 476
/altparam 478
Windows* compiler options *(continued)*
/arch 480
/architecture 480
/asmattr 483
/asmfile 485
/assume 486
/auto 498
/automatic 498
/bigobj 506
/bintext 507
/c 509
/C 511
/CB 511
/ccdefault 510
/check 511
/cm 588
/compile-only 509
/convert 517
/CU 511
/D 522
/d_lines 523, 856
/dbglibs 524
/debug 529
/debug-parameters 532
/define 522
/dll 553
/double-size 554
/E 561
/EP 562
/error-limit 547, 871
/exe 563
/extend-source 565
/extfor 566
/extfpp 567
/extlnk 568
/F 569
/f66 570
/f77rtl 572
/Fa 573
/FA 573
/fast 577
/Fe 563, 581
/FI 588
/fixed 588
/fltconsistency 590
/Fm 593
/Fo 600
/fp 601, 606
/fpconstant 616

Windows* compiler options *(continued)*
/fpe 617
/fpe-all 620
/fpp 625, 889
/fpscomp 627
/FR 638
/free 638
/G2 651
/G2-p9000 651
/G5 653
/G6 653
/G7 653
/GB 653
/Ge 656
/gen-interfaces 657
/Gm 660, 670
/Gs 660
/GS 640, 641, 661
/Gz 662, 670
/heap-arrays 662
/help 663
/homeparams 665
/hotpatch 666
/I 667
/iface 670
/include 667
/inline 674
/intconstant 690
/integer-size 691
/LD 553, 710
/libdir 710
/libs 712
/link 715
/logo 716
/map 720
/MD 726
/MDd 726
/MDsd 712, 728
/MG 1108
/ML 712, 733
/MLd 712, 733
/module 734
/MP 735, 743
/MT 739
/MTd 739
/MW 712
/MWs 712
/names 744
/nbs 486

Windows* compiler options *(continued)*
/noinclude 1116
/O 753
/Ob 680, 758
/object 760
/Od 761
/Og 763
/Op 590
/optimize 753
/Os 800
/Ot 802
/Ox 753
/Oy 598, 600, 803
/P 827
/pdbfile 822
/preprocess-only 827
/prof-func-order 834, 996
/Qansi-alias 479, 847
/Qauto 498
/Qauto_scalar 496, 848
/Qautodouble 1046
/Qax 500, 850
/Qchkstk 853
/Qcommon-args 486
/Qcomplex-limited-range 516, 855
/Qcpp 625, 889
/Qdiag 533, 539, 857, 863
/Qdiag-dump 538, 862
/Qdiag-enable:sc-include 544, 867
/Qdiag-enable:sc-parallel 545, 869
/Qdiag-error-limit 547, 871
/Qdiag-file 548, 872
/Qdiag-file-append 550, 873
/Qdiag-id-numbers 551, 875
/Qdiag-once 552, 876
/Qd-lines 523, 856
/Qdps 478
/Qdyncom 560, 877
/Qextend-source 565
/Qfast-transcendentals 578, 879
/Qfma 593, 880
/Qfnalign 574, 882
/Qfnsplit 597, 883
/Qfpp 625, 889
/Qfp-port 611, 884
/Qfp-relaxed 612, 885
/Qfp-speculation 613, 886
/Qfp-stack-check 615, 888
/Qftz 643, 891

Windows* compiler options *(continued)*
/Qglobal-hoist 658, 893
/QIA64-fr32 894
/QIfist 1008, 1044
/Qimsl 895
/Qinline-debug-info 676, 896
/Qinline-dllimport 897
/Qinline-factor 677, 898
/Qinline-forceinline 679, 900
/Qinline-max-per-compile 682, 901
/Qinline-max-per-routine 683, 903
/Qinline-max-size 685, 905
/Qinline-max-total-size 687, 906
/Qinline-min-size 688, 908
/Qinstruction 730, 911
/Qinstrument-functions 586, 912
/Qip 693, 914
/QIPF-fltacc 698, 919
/QIPF-flt-eval-method0 696, 917
/QIPF-fma 593, 880
/QIPF-fp-relaxed 612, 885
/Qip-no-inlining 694, 915
/Qip-no-pinlining 695, 916
/Qipo 699, 920
/Qipo-c 701, 922
/Qipo-jobs 702, 923
/Qipo-S 704, 925
/Qipo-separate 705, 926
/Qivdep-parallel 707, 927
/Qkeep-static-consts 589, 928
/Qlocation 929
/Qlowercase 744
/Qmap-opts 721, 931
/Qmkl 732, 933
/Qnobss-init 746, 934
/Qonetrip 764, 935
/Qopenmp 765, 936
/Qopenmp-lib 766, 937
/Qopenmp-link 768, 939
/Qopenmp-profile 770, 940
/Qopenmp-report 771, 942
/Qopenmp-stubs 772, 943
/Qopenmp-threadprivate 774, 944
/Qopt-block-factor 775, 946
/Qoption 969
/Qopt-jump-tables 776, 947
/Qopt-loadpair 778, 948
/Qopt-mem-bandwidth 780, 949
/Qopt-mod-versioning 782, 951

Windows* compiler options *(continued)*
/Qopt-multi-version-aggressive 783, 952
/Qopt-prefetch 784, 953
/Qopt-prefetch-initial-values 786, 955
/Qopt-prefetch-issue-excl-hint 787, 956
/Qopt-prefetch-next-iteration 788, 957
/Qopt-ra-region-strategy 790, 959
/Qopt-report 791, 960
/Qopt-report-file 793, 962
/Qopt-report-help 794, 963
/Qopt-report-phase 795, 964
/Qopt-report-routine 796, 965
/Qopt-streaming-stores 797, 966
/Qopt-subscript-in-range 799, 968
/Qpad 806, 971
/Qpad-source 807, 972
/Qpar-adjust-stack 974
/Qpar-affinity 808, 975
/Qparallel 819, 986
/Qpar-num-threads 810, 977
/Qpar-report 811, 978
/Qpar-schedule 814, 980
/Qpar-threshold 818, 984
/Qpc 821, 987
/Qprec 737, 989
/Qprec-div 825, 990
/Qprec-sqrt 826, 991
/Qprof-data-order 829, 992
/Qprof-dir 830, 994
/Qprof-file 832, 995
/Qprof-gen 836, 998
/Qprof-genx 836, 998
/Qprof-hotness-threshold 838, 1000
/Qprof-src-dir 840, 1001
/Qprof-src-root 841, 1003
/Qprof-src-root-cwd 843, 1005
/Qprof-use 845, 1006
/Qrcd 1008, 1044
/Qrct 1009, 1045
/Qsafe-cray-ptr 1010, 1051
/Qsave 1012, 1053
/Qsave-temps 1013, 1054
/Qscalar-rep 1015, 1056
/Qsfalign 1016
/Qsox 1017, 1062
/Qtcheck 1019, 1072
/Qtcollect 1020, 1073
/Qtcollect-filter 1021, 1075
/Qtprofile 1023, 1078

Windows* compiler options *(continued)*
/Qtrapuv 642, 1024
/Qunroll 1026, 1085
/Qunroll-aggressive 1027, 1086
/Quppercase 744
/Quse-asm 1028, 1088
/Quse-msasm-symbols 1029
/Quse-vcdebug 1030
/Qvc 1031
/Qvec 1032, 1090
/Qvec-guard-write 1033, 1091
/Qvec-report 1034, 1092
/Qvec-threshold 1036, 1094
/Qvms 1095
/Qx 1038, 1112
/Qzero 1042, 1121
/real-size 1046
/recursive 1047
/reentrancy 1049
/RTCu 511
/S 1050
/source 1061
/stand 1063
/static 1065
/syntax-only 1070
/Tf 1061
/threads 1077
/traceback 1080
/u 1083
/U 1084
/undefine 1084
/us 486
/V 1090
/vms 1095
/w 1098, 1099
/W0 1099
/W1 1099
/warn 1099
/watch 1105
/WB 1106
/what 1107
/winapp 1108
/X 1116
/Z7 529, 650, 1119, 1122
/Zd 529, 1121
/Zi 529, 650, 1119, 1122
/Zl 710
/Zp 472, 1124
/Zs 1070, 1124
Windows* compiler options *(continued)*
/Zx 1124
Windows* coordinates
functions filling (color index) 2658
functions filling (RGB) 2661
subroutine converting from viewport coordinates 2804
subroutines converting from physical coordinates 2798
Windows* fonts
initializing 2899
Windows* properties
function returning 2799
function setting 3491, 3502
window unit number
function converting to handle 2767
WINEXIT 2345
predefined QuickWin routine 2345
WINFULLSCREEN 2345
predefined QuickWin routine 2345
WININDEX 2345
predefined QuickWin routine 2345
WININPUT 2345
predefined QuickWin routine 2345
WINPASTE 2345
predefined QuickWin routine 2345
WINPRINT 2345
predefined QuickWin routine 2345
WINSAVE 2345
predefined QuickWin routine 2345
WINSELECTALL 2345
predefined QuickWin routine 2345
WINSELECTGRAPHICS 2345
predefined QuickWin routine 2345
WINSELECTTEXT 2345
predefined QuickWin routine 2345
WINSIZETOFIT 2345
predefined QuickWin routine 2345
WINSTATE 2345
predefined QuickWin routine 2345
WINSTATUS 2345
predefined QuickWin routine 2345
WINTILE 2345
predefined QuickWin routine 2345
WINUSING 2345
predefined QuickWin routine 2345
worker thread 1406
working directory
function returning path of 2740

WORKSHARE 1359, 3670
    using 1359
worksharing 1242, 1359, 1360, 1371, 1392, 1447, 3147, 3670
    directives 1359, 1371
WRAPON 3671
WRITE 2120, 3673
    specifier for INQUIRE 2120
write mode
    function returning logical 2805
    function setting logical 3498
write operations
    function committing to physical device 2471

X

X 2071
    edit descriptor 2071
X_floating format 1777
xiar 1504, 1508
xild 1497, 1504, 1508
xilib 1508
xilib.exe file 437
xilibtool 1508
xilink 1497, 1504, 1508
xilink.exe file 437
XOR 2870

Y

year
    subroutine returning 4-digit 2509

Z

ZABS 2321
ZCOS 2486
Z edit descriptor 2048
zero-extend function 3676
zero-size array sections 1807
ZEXP 2647
ZEXT 3676
ZLOG 2979
ZSIN 3511
ZSQRT 3561
ZTAN 3598