

# PGI<sup>®</sup> Fortran Reference

**The Portland Group<sup>®</sup>  
STMicroelectronics  
Two Centerpointe Drive  
Lake Oswego, OR 97035**

While every precaution has been taken in the preparation of this document, The Portland Group® (PGI®), a wholly-owned subsidiary of STMicroelectronics, Inc., makes no warranty for the use of its products and assumes no responsibility for any errors that may appear, or for damages resulting from the use of the information contained herein. The Portland Group retains the right to make changes to this information at any time, without notice. The software described in this document is distributed under license from STMicroelectronics and/or The Portland Group and may be used or copied only in accordance with the terms of the license agreement ("EULA").

No part of this document may be reproduced or transmitted in any form or by any means, for any purpose other than the purchaser's or the end user's personal use without the express written permission of STMicroelectronics and/or The Portland Group.

## PGI® Fortran Reference

Copyright © 2005-2010, STMicroelectronics, Inc.  
All rights reserved.

Printed in the United States of America

First Printing: Release 6.0 March, 2005  
Second Printing: Release 6.1 December, 2005  
Third Printing: Release 6.1-3 February, 2006  
Fourth Printing: Release 7.0-1 December, 2006  
Fifth Printing: Release 7.0-2 February, 2007  
Sixth Printing: Release 7.2-1 May, 2008  
Seventh Printing: Release 7.2-4 August, 2008  
Eighth Printing: Release 8.0-1 November, 2008  
Ninth Printing: Release 8.1 November, 2008  
Tenth Printing: Release 9.0 June, 2009  
Eleventh Printing: Release 2010 November, 2009  
Twelfth Printing: Release 2010, 10.3 March, 2010

[www.pgroup.com](http://www.pgroup.com)

Technical support: [trs@pgroup.com](mailto:trs@pgroup.com)  
Sales: [sales@pgroup.com](mailto:sales@pgroup.com)  
Web: [www.pgroup.com](http://www.pgroup.com)

# Contents

<b>Preface</b> .....	xxiii
Audience Description .....	xxiii
Compatibility and Conformance to Standards .....	xxiii
Organization .....	xxiv
Hardware and Software Constraints .....	xxv
Conventions .....	xxv
Related Publications .....	xxvi
<b>1. Language Overview</b> .....	1
Elements of a Fortran Program Unit .....	1
Statements .....	1
Free and Fixed Source .....	1
Statement Ordering .....	2
The Fortran Character Set .....	3
Free Form Formatting .....	4
Fixed Formatting .....	4
Column Formatting .....	4
Fixed Format Label Field .....	5
Fixed Format Continuation Field .....	5
Fixed Format Statement Field .....	5
Fixed Format Debug Statements .....	5
Tab Formatting .....	6
Fixed Input File Format Summary .....	6
Include Fortran Source Files .....	6
Components of Fortran Statements .....	7
Symbolic Names .....	7
Expressions .....	8
Forming Expressions .....	8
Expression Precedence Rules .....	8
Arithmetic Expressions .....	9
Relational Expressions .....	11
Logical Expressions .....	11
Character Expressions .....	12

Character Concatenation .....	12
Symbolic Name Scope .....	12
Assignment Statements .....	12
Arithmetic Assignment .....	13
Logical Assignment .....	13
Character Assignment .....	14
Listing Controls .....	14
OpenMP Directives .....	15
HPF Directives .....	15
<b>2. Fortran Data Types .....</b>	<b>17</b>
Intrinsic Data Types .....	17
Kind Parameter .....	17
Number of Bytes Specification .....	18
Constants .....	20
Integer Constants .....	20
Binary, Octal and Hexadecimal Constants .....	20
Real Constants .....	21
Double Precision Constants .....	21
Complex Constants .....	22
Double Complex Constants .....	22
Logical Constants .....	22
Character Constants .....	22
PARAMETER Constants .....	23
Derived Types .....	23
Arrays .....	23
Array Declaration Element .....	24
Deferred Shape Arrays .....	24
Subscripts .....	24
Character Substring .....	25
Fortran Pointers and Targets .....	25
Fortran Binary, Octal and Hexadecimal Constants .....	25
Octal and Hexadecimal Constants - Alternate Forms .....	26
Hollerith Constants .....	27
Structures .....	28
Records .....	28
UNION and MAP Declarations .....	29
Data Initialization .....	31
Pointer Variables .....	31
Restrictions .....	32
Pointer Assignment .....	33
<b>3. Fortran Statements .....</b>	<b>35</b>
Statement Format Overview .....	35
Definition of Statement-related Terms .....	35
Origin of Statement .....	35

List-related Notation .....	36
Fortran Statement Summary Table .....	36
ACCEPT .....	41
ALLOCATABLE .....	42
ALLOCATE .....	43
ARRAY .....	44
ASSIGN .....	45
BACKSPACE .....	46
BLOCK DATA .....	46
BYTE .....	47
CALL .....	47
CASE .....	49
CHARACTER .....	49
CLOSE .....	50
COMMON .....	51
COMPLEX .....	54
CONTAINS .....	55
CONTINUE .....	56
CYCLE .....	56
DATA .....	56
DEALLOCATE .....	57
DECODE .....	58
DIMENSION .....	59
DO (Iterative) .....	60
DO WHILE .....	62
DOUBLE COMPLEX .....	62
DOUBLE PRECISION .....	63
ELSE .....	64
ELSE IF .....	65
ELSE WHERE .....	65
ENCODE .....	66
END .....	67
END DO .....	67
END FILE .....	68
END IF .....	68
END MAP .....	69
END SELECT .....	69
END STRUCTURE .....	70
END UNION .....	70
END WHERE .....	70
ENTRY .....	71
EQUIVALENCE .....	73
EXIT .....	74
EXTERNAL .....	74
EXTRINSIC .....	74
FORALL .....	75

FORMAT .....	76
FUNCTION .....	78
GOTO (Assigned) .....	79
GOTO (Computed) .....	80
GOTO (Unconditional) .....	80
IF (Arithmetic) .....	81
IF (Block) .....	81
IF (Logical) .....	82
IMPLICIT .....	82
IMPORT .....	83
INCLUDE .....	84
INQUIRE .....	85
INTEGER .....	88
INTENT .....	89
INTERFACE .....	90
INTRINSIC .....	91
LOGICAL .....	92
MAP .....	93
MODULE .....	94
NAMELIST .....	95
NULLIFY .....	95
OPEN .....	96
OPTIONAL .....	100
OPTIONS .....	100
PARAMETER .....	101
PAUSE .....	102
POINTER .....	102
POINTER (Cray) .....	102
PRINT .....	103
PRIVATE .....	104
PROGRAM .....	105
PUBLIC .....	106
PURE .....	106
READ .....	107
REAL .....	109
RECORD .....	110
RECURSIVE .....	111
REDIMENSION .....	111
RETURN .....	112
REWIND .....	113
SAVE .....	113
SELECT CASE .....	114
SEQUENCE .....	115
STOP .....	115
STRUCTURE .....	116
SUBROUTINE .....	117

TARGET .....	118
THEN .....	118
TYPE .....	119
UNION .....	119
USE .....	121
VOLATILE .....	122
WAIT .....	123
WHERE .....	124
WRITE .....	125
<b>4. Fortran Arrays .....</b>	<b>129</b>
Array Types .....	129
Explicit Shape Arrays .....	130
Assumed Shape Arrays .....	130
Deferred Shape Arrays .....	130
Assumed Size Arrays .....	130
Array Specification .....	130
Explicit Shape Arrays .....	130
Assumed Shape Arrays .....	130
Deferred Shape Arrays .....	131
Assumed Size Arrays .....	131
Array Subscripts and Access .....	131
Array Sections and Subscript Triplets .....	131
Array Sections and Vector Subscripts .....	132
Array Constructors .....	132
CM Fortran Extensions .....	132
The ARRAY Attribute § .....	132
Array Constructors Extensions § .....	133
<b>5. Input and Output .....</b>	<b>135</b>
File Access Methods .....	135
Standard Preconnected Units .....	136
Opening and Closing Files .....	136
Direct Access Files .....	136
Closing a File .....	137
Data Transfer Statements .....	138
Unformatted Data Transfer .....	139
Formatted Data Transfer .....	139
Implied DO List Input Output List .....	139
Format Specifications .....	140
Variable Format Expressions .....	148
Non-advancing Input and Output .....	148
List-directed formatting .....	149
List-directed input .....	149
List-directed output .....	150
Commas in External Field .....	151

Namelist Groups .....	151
Namelist Input .....	151
Namelist Output .....	152
<b>6. Fortran Intrinsic</b> .....	<b>153</b>
Intrinsic Support .....	153
ABS .....	165
ACHAR .....	166
ACOS .....	166
ACOSD .....	166
ADJUSTL .....	167
ADJUSTR .....	167
AIMAG .....	167
AINT .....	168
ALL .....	168
ALLOCATED .....	169
AND .....	169
ANINT .....	169
ANY .....	170
ASIN .....	170
ASIND .....	170
ASSOCIATED .....	171
ATAN .....	171
ATAN2 .....	172
ATAN2D .....	172
ATAND .....	172
BIT_SIZE .....	173
BTEST .....	173
CEILING .....	173
CHAR .....	174
CMPLX .....	174
COMPL .....	175
COMMAND_ARGUMENT_COUNT .....	175
CONJG .....	175
COS .....	176
COSD .....	176
COSH .....	177
COUNT .....	177
CPU_TIME .....	177
CSHIFT .....	178
DATE_AND_TIME .....	178
DBLE .....	179
DCMPLX .....	179
DIGITS .....	180
DIM .....	180
DOT_PRODUCT .....	180



DPROD .....	181
EOSHIFT .....	181
EPSILON .....	182
EQV .....	182
EXP .....	182
EXPONENT .....	183
EXTENDS_TYPE_OF .....	183
FLOOR .....	184
FRACTION .....	184
GET_COMMAND_ARGUMENT .....	184
GET_COMMAND .....	185
GET_ENVIRONMENT_VARIABLE .....	186
HUGE .....	187
IACHAR .....	187
IAND .....	188
IBCLR .....	188
IBITS .....	188
IBSET .....	189
ICHAR .....	189
IEOR .....	189
IINT .....	190
INDEX .....	190
ININT .....	191
INT .....	191
INT8 .....	192
IOR .....	192
ISHFT .....	192
IS_IOSTAT_END .....	193
IS_IOSTAT_EOR .....	193
ISHFTC .....	194
IZEXT .....	195
JINT .....	195
JNINT .....	195
KIND .....	196
KNINT .....	196
LBOUND .....	196
LEADZ .....	197
LEN .....	198
LEN_TRIM .....	198
LGE .....	198
LGT .....	199
LLE .....	199
LLT .....	200
LOC .....	200
LOG .....	200
LOG10 .....	201

LOGICAL .....	201
LSHIFT .....	201
MATMUL .....	202
MAX .....	202
MAXEXPONENT .....	203
MAXLOC .....	203
MAXVAL .....	204
MERGE .....	204
MIN .....	205
MINEXPONENT .....	205
MINLOC .....	205
MINVAL .....	206
MOD .....	207
MODULO .....	207
MOVE_ALLOC .....	207
MVBITS .....	208
NEAREST .....	208
NEQV .....	209
NEW_LINE .....	209
NINT .....	210
NOT .....	210
NULL .....	210
OR .....	211
PACK .....	211
POPCNT .....	211
POPPAR .....	212
PRECISION .....	213
PRESENT .....	213
PRODUCT .....	214
RADIX .....	214
RANDOM_NUMBER .....	214
RANDOM_SEED .....	215
RANGE .....	215
REAL .....	216
REPEAT .....	216
RESHAPE .....	217
RRSPACING .....	217
RSHIFT .....	217
SAME_TYPE_AS .....	218
SCALE .....	218
SCAN .....	218
SELECTED_INT_KIND .....	219
SELECTED_REAL_KIND .....	219
SET_EXPONENT .....	220
SHAPE .....	220
SHIFT .....	221

SIGN .....	221
SIN .....	222
SIND .....	222
SINH .....	222
SIZE .....	223
SPACING .....	223
SPREAD .....	223
SQRT .....	224
SUM .....	224
SYSTEM_CLOCK .....	225
TAN .....	225
TAND .....	226
TANH .....	226
TINY .....	226
TRANPOSE .....	227
TRIM .....	227
UBOUND .....	227
UNPACK .....	228
VERIFY .....	228
XOR .....	229
ZEXT .....	229
Intrinsic Modules .....	230
Module IEEE_ARITHMETIC .....	230
Module IEEE_EXCEPTIONS .....	234
Module iso_c_binding .....	236
Module iso_fortran_env .....	237
<b>7. 3F Functions and VAX Subroutines .....</b>	<b>239</b>
3F Routines .....	239
abort .....	240
abort .....	240
access .....	240
alarm .....	241
Bessel functions .....	241
chdir .....	242
chmod .....	242
ctime .....	242
date .....	242
error functions .....	243
etime, dtime .....	243
exit .....	243
fdate .....	244
fgetc .....	244
flush .....	244
fork .....	244
fputc .....	245

free	245
fseek	245
ftell	246
gerror	246
getarg	246
iargc	246
getc	247
getcwd	247
getenv	247
getgid	248
getlog	248
getpid	248
getuid	248
gmtime	248
hostnm	249
idate	249
ierrno	249
ioinit	249
isatty	250
itime	250
kill	250
link	251
lnblk	251
loc	251
ltime	251
malloc	252
mclock	252
mvbits	252
oustr	252
perror	253
putc	253
putenv	253
qsort	253
rand, irand, srand	254
random, irandm, drandm	254
range	255
rename	255
rindex	256
secnds, dsecnds	256
setvbuf	256
setvbuf3f	257
signal	258
sleep	258
stat, lstat, fstat, fstat64	259
stime	259
symlink	259

system .....	260
time .....	260
times .....	260
ttynam .....	261
unlink .....	261
wait .....	261
VAX System Subroutines .....	261
Built-In Functions .....	262
VAX/VMS System Subroutines .....	262
<b>8. OpenMP Directives for Fortran .....</b>	<b>267</b>
OpenMP Overview .....	267
OpenMP Shared-Memory Parallel Programming Model .....	267
Terminology .....	268
OpenMP Example .....	269
Task Overview .....	270
Tasks .....	270
Task Characteristics and Activities .....	271
Task Scheduling Points .....	271
Task Construct .....	272
Parallelization Directives .....	273
Directive Recognition .....	274
Directive Clauses .....	274
COLLAPSE (n) .....	277
COPYIN (list) .....	277
COPYPRIVATE(list) .....	277
DEFAULT .....	278
FIRSTPRIVATE(list) .....	278
IF() .....	278
LASTPRIVATE(list) .....	278
NOWAIT .....	278
NUM_THREADS .....	278
ORDERED .....	279
PRIVATE .....	279
REDUCTION .....	279
SCHEDULE .....	280
SHARED .....	280
UNTIED .....	280
Directive Summary Table .....	281
ATOMIC .....	282
BARRIER .....	282
CRITICAL ... END CRITICAL .....	283
C\$DOACROSS .....	284
DO...END DO .....	284
FLUSH .....	286
MASTER ... END MASTER .....	286

ORDERED .....	287
PARALLEL ... END PARALLEL .....	288
PARALLEL DO .....	289
PARALLEL SECTIONS .....	290
PARALLEL WORKSHARE .....	290
SECTIONS ... END SECTIONS .....	291
SINGLE ... END SINGLE .....	292
TASK .....	292
TASKWAIT .....	294
THREADPRIVATE .....	294
WORKSHARE ... END WORKSHARE .....	295
Runtime Library Routines .....	296
OpenMP Environment Variables .....	300
OMP_DYNAMIC .....	301
OMP_NESTED .....	301
OMP_MAX_ACTIVE_LEVELS .....	301
OMP_NUM_THREADS .....	301
OMP_SCHEDULE .....	302
OMP_STACKSIZE .....	302
OMP_THREAD_LIMIT .....	302
OMP_WAIT_POLICY .....	302
<b>9. Interoperability with C .....</b>	<b>305</b>
Enumerators .....	305
Interoperability with C Pointer Types .....	305
c_f_pointer .....	305
c_f_procpointer .....	307
c_associated .....	308
Interoperability of Derived Types .....	308
<b>10. HPF .....</b>	<b>311</b>
Supported HPF Intrinsic .....	311
CM Fortran Intrinsic .....	313
CSHIFT .....	313
EOSHIFT .....	313
RESHAPE .....	314
HPF Directives .....	314
Adding HPF Directives to Programs .....	314
HPF Directive Summary .....	315
ALIGN - REALIGN .....	316
DIMENSION .....	317
DYNAMIC .....	318
DISTRIBUTE - REDISTRIBUTE .....	318
INDEPENDENT .....	319
INHERIT .....	320
PROCESSORS .....	320

NO SEQUENCE .....	321
SEQUENCE .....	321
TEMPLATE .....	322
HPF_LOCAL .....	322
ABSTRACT_TO_PHYSICAL .....	324
GLOBAL_ALIGNMENT .....	324
GLOBAL_DISTRIBUTION .....	324
GLOBAL_LBOUND .....	324
GLOBAL_SHAPE .....	325
GLOBAL_SIZE .....	325
GLOBAL_TEMPLATE .....	326
GLOBAL_TO_LOCAL .....	326
GLOBAL_UBOUND .....	327
LOCAL_BLKCNT .....	328
LOCAL_LINDEX .....	328
LOCAL_TO_GLOBAL .....	329
LOCAL_UINDEX .....	329
MY_PROCESSOR .....	330
PHYSICAL_TO_ABSTRACT .....	330
<b>Index .....</b>	<b>333</b>





# Figures

1.1. Order of Statements .....	2
--------------------------------	---



# Tables

1.1. Fortran Characters .....	3
1.2. C Language Character Escape Sequences .....	3
1.3. Fixed Format Record Positions and Fields .....	5
1.4. Fortran Operator Precedence .....	8
1.5. Arithmetic Operators .....	10
1.6. Arithmetic Operator Precedence .....	10
1.7. Relational Operators .....	11
1.8. Logical Expression Operators .....	11
2.1. Fortran Intrinsic Data Types .....	17
2.2. Data Types Kind Parameters .....	18
2.3. Data Type Extensions .....	18
2.4. Data Type Ranks .....	19
2.5. Examples of Real Constants .....	21
2.6. Examples of Double Precision Constants .....	21
3.1. Intrinsic Summary Table .....	36
3.2. OPTIONS Statement .....	100
5.1. OPEN Specifiers .....	137
5.2. Format Character Controls for a Printer .....	141
5.3. List Directed Input Values .....	149
5.4. Default List Directed Output Formatting .....	150
6.1. Fortran 90/95 Bit Manipulation Functions and Subroutines .....	153
6.2. Elemental Character and Logical Functions .....	154
6.3. Fortran 90/95 Vector/Matrix Functions .....	156
6.4. Fortran 90/95 Array Reduction Functions .....	156
6.5. Fortran 90/95 String Construction Functions .....	157
6.6. Fortran 90/95 Array Construction/Manipulation Functions .....	157
6.7. Fortran 90/95 General Inquiry Functions .....	158
6.8. Fortran 90/95 Numeric Inquiry Functions .....	158
6.9. Fortran 90/95 Array Inquiry Functions .....	159
6.10. Fortran 90/95 Subroutines .....	159
6.11. Fortran 90/95 Transfer Function .....	160
6.12. Arithmetic Functions .....	160
6.13. Fortran 2003 Functions .....	164

6.14. Miscellaneous Functions .....	165
6.15. IEEE_ARITHMETIC Derived Types .....	231
6.16. IEEE_ARITHMETIC Inquiry Functions .....	231
6.17. IEEE_ARITHMETIC Elemental Functions .....	232
6.18. IEEE_ARITHMETIC Non-Elemental Subroutines .....	234
6.19. IEEE_EXCEPTIONS Derived Types .....	235
6.20. IEEE_EXCEPTIONS Inquiry Functions .....	235
6.21. IEEE_EXCEPTIONS Elemental Subroutines .....	236
6.22. IEEE_EXCEPTIONS Elemental Subroutines .....	236
6.23. iso_fortran_env Named Constants .....	237
8.1. Directive Clauses Summary Table .....	274
8.2. Initialization of REDUCTION Variables .....	279
8.3. Directive Summary Table .....	281
8.4. Run-time Library Routines Summary .....	296
8.5. OpenMP-related Environment Variable Summary Table .....	300
10.1. HPF Intrinsic and Library Procedures .....	311
10.2. HPF Directive Summary .....	315
10.3. HPF_LOCAL_LIBRARY Procedures .....	323

# Examples

8.1. OpenMP Loop Example .....	269
8.2. OpenMP Task Fortran Example .....	272
9.1. Enumerator Example .....	305
9.2. Derived Type Interoperability .....	309



# Preface

This manual describes The Portland Group's implementation of the FORTRAN 77 and Fortran 90/95 as well as High Performance Fortran (HPF) languages. Collectively, The Portland Group compilers that implement these languages are referred to as the PGI Fortran compilers. This manual is part of a set of documents describing the Fortran language and the compilation tools available from The Portland Group. It presents the Fortran language statements, intrinsics, and extension directives. The Portland Group's Fortran compilation system includes a compilation driver, multiple Fortran compilers, associated runtime support and mathematical libraries, and associated software development tools for debugging and profiling the performance of Fortran programs. Depending on the target system, The Portland Group's Fortran software development tools may also include an assembler or a linker. You can use these tools to create, debug, optimize and profile your Fortran programs. [“Related Publications,”](#) on page xxvi lists other manuals in the PGI documentation set.

## Audience Description

This manual is intended for people who are porting or writing Fortran programs using the PGI Fortran compilers. To use Fortran you should be aware of the role of Fortran and of source-level programs in the software development process and you should have some knowledge of a particular system or workstation cluster. To use the PGI Fortran compilers, you need to be familiar with the Fortran language FORTRAN77, Fortran 90/95, or HPE, as well as the basic commands available on your host system.

## Compatibility and Conformance to Standards

The PGI Fortran compilers, PGF77, PGF95 and PGHPE, run on a variety of 32-bit and 64-bit x86 processor-based host systems. The PGF77 compiler accepts an enhanced version of FORTRAN 77 that conforms to the ANSI standard for FORTRAN 77 and includes various extensions from VAX/VMS Fortran, IBM/VS Fortran, and MIL-STD-1753. The PGF95 compiler accepts a similarly enhanced version of the ANSI standard for Fortran 90/95. The PGHPF compiler accepts the HPF language and is largely, though not strictly, a superset of Fortran 90/95. The PGHPF compiler conforms to the High Performance Fortran Language Specification Version 1.1, published by the Center for Research on Parallel Computation, at Rice University (with a few limitations and modifications, consult the PGHPF Release Notes for details).

For further information on the Fortran language, you can also refer to the following:

- American National Standard Programming Language FORTRAN, ANSI X3. -1978 (1978).
- ISO/IEC 1539 : 1991, Information technology – Programming Languages – Fortran, Geneva, 1991 (Fortran 90).
- ISO/IEC 1539 : 1997, Information technology – Programming Languages – Fortran, Geneva, 1997 (Fortran 95).
- Fortran 95 Handbook Complete ISO/ANSI Reference, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- High Performance Fortran Language Specification, Revision 1.30 or 1.1, Rice University, Houston, Texas (1993), <http://www.crpc.rice.edu/HPFF>.
- High Performance Fortran Language Specification, Revision 2.0, Rice University, Houston, Texas (1997), <http://www.crpc.rice.edu/HPFF>.
- OpenMP Fortran Application Program Interface, Version 2.0, November 1999, <http://www.openmp.org>.
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).

## Organization

This manual is divided into the following chapters and appendices:

[Chapter 1, “\*Language Overview\*”](#), provides an introduction to the Fortran language.

[Chapter 2, “\*Fortran Data Types\*”](#), describes the data types supported by PGI Fortran compilers and provides examples using various data types. It also contains information on memory allocation and alignment issue.

[Chapter 3, “\*Fortran Statements\*”](#), describes each Fortran and HPF statement that the PGI Fortran compilers accept. Many HPF statements are in the form of compiler directives which are ignored by non-HPF compilers.

[Chapter 4, “\*Fortran Arrays\*”](#), describes special characteristics of arrays in Fortran 90/95.

[Chapter 5, “\*Input and Output\*”](#), describes the input, output, and format statements that allow programs to transfer data to or from files.

[Chapter 6, “\*Fortran Intrinsic\*”](#), lists the Fortran intrinsics and subroutines supported by the PGI Fortran compilers.

[Chapter 7, “\*3F Functions and VAX Subroutines\*”](#), describes the functions and subroutines in the Fortran runtime library and discusses the VAX/VMS system subroutines and the built-in functions supported by the PGI Fortran compilers.

[Chapter 8, “\*OpenMP Directives for Fortran\*”](#), lists the language extensions that the PGI Fortran compilers support.



Chapter 9, “*Interoperability with C*”, describes the pointer types and enumerators available for Fortran interoperability with C.

Chapter 10, “*HPF*”, describes the High Performance Fortran intrinsics, the HPF directives which support data distribution and alignment, and influence data parallelism by providing additional information to the PGHPF compiler, and lists the HPF\_LOCAL\_LIBRARY procedures supported by the PGHPF compiler.

## Hardware and Software Constraints

The PGI compilers operate on a variety of host systems and produce object code for a variety of target systems. Details concerning environment-specific values and defaults and host-specific features or limitations are presented in the PGI User’s Guide, the man pages for each compiler in a given installation, and in the release notes and installation instructions included with all PGI compilers and tools software products.

## Conventions

This PGI Fortran Reference manual uses the following conventions:

*italic*

is used for commands, filenames, directories, arguments, options and for emphasis.

*Constant Width*

is used in examples and for language statements in the text.

[item]

square brackets indicate optional items. In this case item is optional.

{ item2 | item3 }

braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...

ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN

Fortran language statements are shown using upper-case characters and a reduced point size.

<TAB>

non-printing characters, such as TAB, are shown enclosed in greater than and less than characters and a reduced point size.

§

this symbol indicates an area in the text that describes a Fortran 90/95 Language enhancement. Enhancements are features that are not described in the ANSI Fortran 90/95 standards.

@

This symbol indicates an area in the text that describes a FORTRAN 77 enhancement. Enhancements may be VAX/VMS, IBM/VM, or military standard MIL-STD-1753 enhancements.

## Related Publications

The following documents contain additional information related to HPF and other compilers and tools available from The Portland Group, Inc.

- The PGI User's Guide describes the general features and usage guidelines for all PGI compilers, and describes in detail various available compiler options in a user's guide format.
- The PGHPF User's Guide describes the PGHPF compiler and describes some details concerning the PGI implementation of HPF in a user's guide format.
- Fortran 95 Handbook, from McGraw-Hill, describes the Fortran 95 language and the statements, data types, input/output format specifiers, and additional reference material that defines ANSI/ISO Fortran 95.
- System V Application Binary Interface Processor Supplement by AT&T UNIX System Laboratories, Inc, (available from Prentice Hall, Inc.)
- The High Performance Fortran Handbook, from MIT Press, describes the HPF language in detail.
- High Performance Fortran Language Specification, Rice University, Houston Texas (1993), is the specification for the HPF language and is available online at <http://www.crpc.rice.edu/HPFF>.
- American National Standard Programming Language Fortran, ANSI x.3-1978 (1978).
- Programming in VAX FORTRAN, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS FORTRAN, IBM Corporation, Rev. GC26-4119.
- Military Standard, FORTRAN, DOD Supplement to American National Standard Programming Language FORTRAN, ANSI X3.-1978, MIL-STD-1753 (November 9, 1978).

# Chapter 1. Language Overview

This chapter describes the basic elements of the Fortran language, the format of Fortran statements, and the types of expressions and assignments accepted by the PGI Fortran compilers.

The PGF77 compiler accepts as input FORTRAN 77 and produces as output assembly language code, binary object code or binary executables in conjunction with the assembler, linker and libraries on the target system. The input language must be extended FORTRAN 77 as specified in this reference manual. The PGF95 and PGHPF compilers function similarly for Fortran 90/95 and HPF respectively.

This chapter is not an introduction to the overall capabilities of Fortran. Rather, it is an overview of the syntax requirements of programs used with the PGI Fortran compilers. The Fortran 95 Handbook and The High Performance Fortran Handbook provide details on the capabilities of Fortran 90/95 and HPF languages.

## Elements of a Fortran Program Unit

A Fortran program is composed of SUBROUTINE, FUNCTION, MODULE, BLOCK DATA, or PROGRAM program units.

Fortran source code consists of a sequence of program units which are to be compiled. Every program unit consists of statements and optionally comments beginning with a program unit statement, either a SUBROUTINE, FUNCTION, or PROGRAM statement, and finishing with an END statement (BLOCK DATA and MODULE program units are also allowed).

In the absence of one of these statements, the PGI Fortran compilers insert a PROGRAM statement.

## Statements

Statements are either executable statements or nonexecutable specification statements. Each statement consists of a single line or source record, possibly followed by one or more continuation lines. Multiple statements may appear on a single line if they are separated by a semicolon (;). Comments may appear on any line following a comment character (!).

## Free and Fixed Source

Fortran permits two types of source formatting, fixed source form and free source form.

- **Fixed source form** uses the traditional Fortran approach where specific column positions are reserved for labels, continuation characters, and statements and blank characters are ignored. The PGF77 compiler supports only fixed source form. The PGF77 compiler also supports a less restrictive variety of fixed source form called tab source form.
- **Free source form** introduced with Fortran 90 places few restrictions on source formatting; the context of an element, as well as the position of blanks, or tabs, separate logical tokens. You can select free source form as an option to PGF95 or PGHPF in one of these ways:
  - Use the compiler option `-Mfreeform`.
  - Use either the suffix `.f90` or the suffix `.f95`.

## Statement Ordering

Fortran statements and constructs must conform to ordering requirements imposed by the language definition. [Figure 1.1, “Order of Statements”](#) illustrates these requirements. Vertical lines separate statements and constructs that can be interspersed. Horizontal lines separate statements that must not be interspersed.

These rules are less strict than those in the ANSI standard. The differences are as follows:

- DATA statements can be freely interspersed with PARAMETER statements and other specification statements.
- NAMELIST statements are supported and have the same order requirements as FORMAT and ENTRY statements.
- The IMPLICIT NONE statement can precede other IMPLICIT statements.

Figure 1.1. Order of Statements

Comments and INCLUDE Statements	OPTIONS Statement			
	PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statements			
	USE Statements			
	IMPORT Statements			
	NAMELIST, FORMAT, and ENTRY Statements	IMPLICIT NONE Statements		
		IMPLICIT Statements		PARAMETER
		Data Statements	Other Specifications	
			Statement Function Definition	
			EXECUTABLE Statements	
	CONTAINS Statement			
Internal Subprograms or Module				
END Statement				

## The Fortran Character Set

Table 1.1, “Fortran Characters”, shows the set of Fortran characters. Character variables and constants can use any ASCII character. The value of the command-line option `-Mupcase` determines if the compiler distinguishes between case (upper and lower) in identifiers. By default, without the `-Mupcase` option selected, the compiler does not distinguish between upper and lower case characters in identifiers (upper and lower case are always significant in character constants).

Table 1.1. Fortran Characters

Character	Description	Character	Description
,	Comma	A-Z, a-z	Alphabetic
:	Colon	<space>	Space character
;	Semicolon	=	Equals
_	Underscore character	+	Plus
<	Less than	-	Minus
>	Greater than	*	Asterisk
?	Question mark	/	Slash
%	Percent	(	Left parenthesis
"	Quotation mark	)	Right parenthesis
\$	Currency symbol	[	Left bracket
.	Decimal point	]	Right bracket
!	Exclamation mark	<CR>	Carriage return
0-9	Numeric	<TAB>	Tabulation character

Table 1.2, “C Language Character Escape Sequences”, shows C language character escape sequences that the PGI Fortran compilers recognize in character string constants when `-Mbackslash` is on the command line. These values depend on the command-line option `-Mbackslash`.

Table 1.2. C Language Character Escape Sequences

Character	Description
\v	vertical tab
\a	alert (bell)
\n	newline
\t	tab
\b	backspace
\f	formfeed
\r	carriage return
\0	null
\'	apostrophe (does not terminate a string)

Character	Description
\"	double quotes (does not terminate a string)
\\	\
\x	x, where x is any other character
\ddd	character with the given octal representation.

## Free Form Formatting

Using free form formatting, columns are not significant for the elements of a Fortran line, and a blank or series of blanks or tabs and the context of a token specify the token type. The following rules apply to free form formatting:

- Up to 132 characters are valid per line, and the compiler option `-Mextend` does not apply.
- A single Fortran line may contain multiple statements, with the `;` (semicolon) separating multiple statements on a single line.
- Free format labels are valid at the start of a line.
  - The label must be separated from the remaining statements on the line by at least one blank or a `<TAB>`.
  - Labels consist of a numeric field drawn from digits 0 to 9.
  - The label cannot be more than 5 characters.
- Either a blank line, or the `!` character following a Fortran line indicates a comment. The Fortran text does not contain any of the characters after the `!`.
- The `&` character at the end of a line means the following line represents a continuation line.
  - If a continuation line starts with the `&` character, then the characters following the `&` are the start of the continuation line.
  - If the continuation line does not start with a `&`, then all characters on the line are part of the continuation line, including any initial blanks or tabs.

A single Fortran line may contain multiple statements. The `;` (semicolon) separates multiple statements on a single line. Free format labels are valid at the start of a line, as long as the label is separated from the remaining statements on the line by at least one blank or a `<TAB>`. Labels consist of a numeric field drawn from digits 0 to 9. The label cannot be more than 5 characters.

## Fixed Formatting

This section describes the two types of fixed formatting that PGI Fortran compilers support: column formatting and tab formatting.

### Column Formatting

When using column formatting a Fortran record consists of a sequence of up to 72 or 132 ASCII characters, the last being `<CR>`. [Table 1.3](#) shows the fixed layout.

## Note

---

For column formatting of 132 characters, you must specify `-Mextend`.

Table 1.3. Fixed Format Record Positions and Fields

Position	Field
1-5	Label field
6	Continuation field
7-72 or 7-132	Statement field

Characters on a line beyond position 72, or position 132 if `-Mextend` is specified, are ignored. In addition, any characters following an exclamation (!) character are considered comments and are thus disregarded during compilation.

### Fixed Format Label Field

The label field holds up to five characters. Further, each label must be unique in its program unit.

- The characters C, D, \*, or ! in the first character position of a label field indicate a comment line.
- When a numeric field drawn from digits 0 to 9 is placed in the label field, the field is a label.
- A line with no label, and with five space characters or a `<TAB>` in the label field, is an unlabeled statement.
- Continuation lines must not be labeled.
- A program to only jump to labels that are on executable statements.

### Fixed Format Continuation Field

The sixth character position, or the position after the tab, is the continuation field. This field is ignored in comment lines. It is invalid if the label field is not five spaces. A value of 0, `<space>` or `<TAB>` indicates the first line of a statement. Any other value indicates a subsequent, continuation line to the preceding statement.

### Fixed Format Statement Field

The statement field consists of valid identifiers and symbols, possibly separated by `<space>` or `<TAB>` and terminated by `<CR>`.

Within the statement field, tabs, spaces, comments and any characters found beyond the 72nd character, or position 132 if `-Mextend` is specified, are ignored. As stated earlier, any characters following an exclamation (!) character are considered comments.

### Fixed Format Debug Statements

The letter D in column 1 using fixed formatting designates the statement on the specified line is a debugging statement. The compiler treats the debugging statement as a comment, ignoring it, unless the command line option `-Mdlines` is set during compilation. If `-Mdlines` is set, the compiler acts as if the line starting with D were a Fortran statement and compiles the line according to the standard rules.

## Tab Formatting

The PGI Fortran compilers support an alternate form of fixed source form called tab source form. A tab formatted source file is made up of a label field, an optional continuation indicator and a statement field. The label field is terminated by a tab character. The label cannot be more than 5 characters.

A continuation line is indicated by a tab character followed immediately by a non-zero digit. The statement field starts after a continuation indicator, when one is present. Again, any characters found beyond the 72nd character, or position 132 if `-Mextend` is specified, are ignored.

## Fixed Input File Format Summary

For fixed input file format, the following is true:

- Tab-Format lines are supported.
  - A tab in columns 1-6 ends the statement label field and begins an optional continuation indicator field.
  - If a non-zero digit follows the tab character, the continuation field exists and indicates a continuation field.
  - If anything other than a non-zero digit follows the tab character, the statement body begins with that character and extends to the end of the source statement.

### Note

---

Note that this does not override Fortran's free source form handling since no valid Fortran statement can begin with a non-zero digit.

- The tab character is ignored if it occurs in a line except in Hollerith or character constants.
- Input lines may be of varying lengths.
  - If there are fewer than 72 characters, the line is padded with blanks.
  - Characters after the 72nd are ignored unless the `-Mextend` option is used on the command line.

### Note

---

The `-Mextend` option extends the statement field to position 132.

When the `-Mextend` option is used, the input line is padded with blanks if it is fewer than 132 characters; characters after the 132nd are ignored.

- Blank lines are allowed at the end of a program unit.
- The number of continuation lines allowed is extended to 1000 lines.

## Include Fortran Source Files

The sequence of consecutive compilation of source statements may be interrupted so that an extra source file can be included. To do this, use the `INCLUDE` statement which takes the form:



```
INCLUDE
"filename"
```

where filename is the name of the file to be included. Pairs of either single or double quotes are acceptable enclosing filename.

The INCLUDE file is compiled to replace the INCLUDE statement, and on completion of that source the file is closed and compilation continues with the statement following the INCLUDE.

INCLUDE files are especially recommended when the same COMMON blocks and the same COMMON block data mappings are used in several program units. For example the following statement includes the file MYFILE.DEF

```
INCLUDE "MYFILE.DEF"
```

Nested includes are allowed, up to a PGI Fortran defined limit of 20.

Recursive includes are not allowed. That is, if a file includes a file, that file may not also include the same file.

## Components of Fortran Statements

Fortran program units are made up of statements which consist of expressions and elements. An expression can be broken down to simpler expressions and eventually to its elements combined with operators. Hence the basic building block of a statement is an element.

An element takes one of the following forms:

- A constant represents a fixed value.
- A variable represents a value which may change during program execution.
- An array is a group of values that can be referred to as a whole, as a section, or separately. The separate values are known as the elements of the array. The array has a symbolic name.
- A function reference or subroutine reference is the name of a function or subroutine followed by an argument list. The reference causes the code specified at function/subroutine definition to be executed and if a function, the result is substituted for the function reference.

## Symbolic Names

Symbolic names identify different entities in Fortran source code. A symbolic name is a string of letters and digits, which must start with a letter and be terminated by a character not in the symbolic names set (for example a <space> or a <TAB> character). Underscore ( \_ ) characters may appear within symbolic names. Only the first thirty-one characters identify the symbolic name.

Here several examples of symbolic names:

```
NUM
CRA9
numericabcdefghijklmnopqrstuvwxy
```

The last example is identified by its first 31 characters and is equivalent to:

```
numericabcdefghijklmnopqrstuvw
```

Some examples of invalid symbolic name include:

8Q Invalid because it begins with a number

FIVE.4 Invalid because it contains a period, an invalid character for a symbolic name.

## Expressions

Each data item, such as a variable or a constant, represents a particular value at any point during program execution. These elements may be combined together to form expressions, using binary or unary operators, so that the expression itself yields a value. A Fortran expression may be any of the following:

- A scalar expression
- An array expression
- A constant expression
- A specification expression
- An initialization expression
- Mixed array and scalar expressions

## Forming Expressions

Expressions fall into one of four classes: arithmetic, relational, logical or character, each class described later in this chapter.

An expression is formed like this:

`expr binary-operator expr` or `unary-operator expr`

where `expr` is formed as an expression or as an element.

For example, these are simple expressions whose components are elements. The first expression involves a binary operator and the other two are unary operators.

`A+B`

`-C`

`+D`

## Expression Precedence Rules

Arithmetic, relational and logical expressions may be identified to the compiler by the use of parentheses, as described in [“Arithmetic Expressions,” on page 9](#). When no guidance is given to the compiler it imposes a set of precedence rules to identify each expression uniquely. [Table 1.1, “Fortran Characters”](#), shows the operator precedence rules for expressions.

Table 1.4. Fortran Operator Precedence

Operator	Evaluated
Unary defined	Highest
**	N/A
* or /	N/A
Unary + or -	N/A

Operator	Evaluated
Binary + or -	N/A
Relational operators: .GT., .GE., .LE.	N/A
Relational operators ==, /=	Same precedence
Relational operators <, <=, >, >=	Same precedence
Relational operators .EQ., .NE., .LT.	Same precedence
.NOT.	N/A
.AND.	N/A
.OR.	N/A
.NEQV. and .EQV.	N/A
Binary defined	Lowest

For example, the following two expressions are equivalent. If we set A to 16, B to 4, and C to 2, both expressions equal 8.

`A/B*C`      such as `16 / 4 * 2`

`(A/B)*C`      such as `(16 / 4) * 2`

Another example of equivalent expressions are these:

`A*B+B*C`      .EQ.      `X+Y/Z`      .AND.      .NOT.      `K-3.0`      .GT.      `T`

`(( (A*B)+(B*C) ) .EQ. (X+(Y/Z))) .AND. (.NOT. ((K-3.0) .GT. T))`

## Arithmetic Expressions

Arithmetic expressions are formed from arithmetic elements and arithmetic operators.

### Arithmetic Elements

An arithmetic element may be:

- an arithmetic expression
- a variable
- a constant
- an array element
- a function reference
- a field of a structure

### Note

A value should be associated with a variable or array element before it is used in an expression.

### Arithmetic Operators

The arithmetic operators specify a computation to be performed on the elements. The result is a numeric result. [Table 1.5](#) shows the arithmetic operators.

Table 1.5. Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
-	Subtraction or unary minus

### Arithmetic Operator Precedence

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. [Table 1.6](#) shows the precedence of each arithmetic operator.

Table 1.6. Arithmetic Operator Precedence

Operator	Precedence
**	First
* and /	Second
+ and -	Third

The following example is resolved into the arithmetic expressions  $(A) + (B * C)$  rather than  $(A + B) * (C)$ .

```
A + B * C
```

Normal ranked precedence may be overcome using parentheses which force the item(s) enclosed to be evaluated first. For example, in the following expression the computer firsts adds A and B, and then multiplies that sum by C.

```
(A + B) * C
```

### Arithmetic Expression Types

The type of an arithmetic expression depends on the type of elements in the expression:

#### INTEGER

if it contains only integer elements.

#### REAL

if it contains only real and integer elements.

#### DOUBLE PRECISION

if it contains only double precision, real and integer elements.

#### COMPLEX

if any element is complex. Any element which needs conversion to complex will be converted by taking the real part from the original value and setting the imaginary part to zero.

**DOUBLE COMPLEX**

if any element is double complex.

Table 2.4, “Data Type Ranks” provides more information about these expressions.

**Relational Expressions**

A relational expression is composed of two arithmetic expressions separated by a relational operator. The value of the expression is true or false (*.TRUE.* or *.FALSE.*) depending on the value of the expressions and the nature of the operator. Table 1.7 shows the relational operators.

Table 1.7. Relational Operators

Operator	Relationship
<i>.LT.</i>	Less than
<i>.LE.</i>	Less than or equal to
<i>.EQ.</i>	Equal to
<i>.NE.</i>	Not equal to
<i>.GT.</i>	Greater than
<i>.GE.</i>	Greater than or equal to

In relational expressions the arithmetic elements are evaluated to obtain their values. The relationship is then evaluated to obtain the true or false result. Thus the relational expression:

```
TIME + MEAN .LT. LAST
```

means if the sum of *TIME* and *MEAN* is less than the value of *LAST*, then the result is true, otherwise it is false.

**Logical Expressions**

A logical expression is composed of two relational or logical expressions separated by a logical operator. Each logical expression yields the value true or false (*.TRUE.* or *.FALSE.*). Table 1.8 shows the logical operators.

Table 1.8. Logical Expression Operators

Operator	Relationship
<i>.AND.</i>	True if both expressions are true.
<i>.OR.</i>	True if either expression or both is true.
<i>.NOT.</i>	This is a unary operator; it is true if the expression is false, otherwise it is false.
<i>.NEQV.</i>	False if both expressions have the same logical value
<i>.XOR.</i>	Same as <i>.NEQV.</i>
<i>.EQV.</i>	True if both expressions have the same logical value

In the following example, *TEST* will be *.TRUE.* if *A* is greater than *B* or *I* is not equal to *J+17*.

```
TEST = A .GT. B .OR. I .NE. J+17
```

## Character Expressions

An expression of type CHARACTER can consist of one or more printable characters. Its length is the number of characters in the string. Each character is numbered consecutively from left to right beginning with 1. For example:

```
'ab_&'
'A@HJi2'
'var[1,12]'
```

## Character Concatenation

A character expression can be formed by concatenating two (or more) valid character expressions using the concatenation operator //. The following table shows several examples of concatenation.

Expression	Value
'ABC'//YZ'	"ABCYZ"
'JOHN '//SMITH'	"JOHN SMITH"
'J'//JAMES '//JOY'	"JJAMESJOY"

## Symbolic Name Scope

Fortran 90/95 and HPF scoping is expanded from the traditional FORTRAN 77 capabilities which provide a scoping mechanism using subroutines, main programs, and COMMONs. Fortran 90/95 and HPF add the MODULE statement. Modules provide an expanded alternative to the use of both COMMONs and INCLUDE statements. Modules allow data and functions to be packaged and defined as a unit, incorporating data hiding and using a scope that is determined with the USE statement.

Names of COMMON blocks, SUBROUTINES and FUNCTIONS are global to those modules that reference them. They must refer to unique objects, not only during compilation, but also in the link stage.

The scope of names other than these is local to the module in which they occur, and any reference to the name in a different module will imply a new local declaration. This includes the arithmetic function statement.

## Assignment Statements

A Fortran assignment statement can be any of the following:

- An intrinsic assignment statement
- A statement label assignment
- An array assignment
- A masked array assignment
- A pointer assignment
- A defined assignment

## Arithmetic Assignment

The arithmetic assignment statement has the following form:

```
object = arithmetic-expression
```

where *object* is one of the following:

- Variable
- Function name (within a function body)
- Subroutine argument
- Array element
- Field of a structure

The type of *object* determines the type of the assignment (INTEGER, REAL, DOUBLE PRECISION or COMPLEX) and the arithmetic-expression is coerced into the correct type if necessary.

In the case of:

```
complex = real expression
```

the implication is that the real part of the complex number becomes the result of the expression and the imaginary part becomes zero. The same applies if the expression is double precision, except that the expression will be coerced to real.

The following are examples of arithmetic assignment statements.

```
A = (P+Q) * (T/V)
B = R**T**2
```

## Logical Assignment

The logical assignment statement has the following form:

```
object = logical-expression
```

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- A field of a structure

The type of *object* must be logical.

In the following example, *FLAG* takes the logical value *.TRUE.* if *P+Q* is greater than *R*; otherwise *FLAG* has the logical value *.FALSE.*

```
FLAG = (P+Q) .GT. R
```

## Character Assignment

The form of a character assignment is:

```
object = character expression
```

where *object* must be of type character, and is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- Character substring
- A field of a structure

In addition, these rules apply:

- None of the character positions being defined in *object* can be referenced in the character expression.
- Only such characters as are necessary for the assignment to *object* need to be defined in the character expression.
- The character expression and *object* can have different lengths.
  - When *object* is longer than the character expression, trailing blanks are added to the object.
  - If *object* is shorter than the character expression the right-hand characters of the character expression are truncated as necessary.

In the following example, all the variables and arrays are assumed to be of type character.

```
FILE = 'BOOKS'  
PLOT(3:8) = 'PLANTS'  
TEXT(I,K+1)(2:B-1) = TITLE//X
```

## Listing Controls

The PGI Fortran compilers recognize three compiler directives that affect the program listing process:

**%LIST**

Turns on the listing process beginning at the following source code line.

**%NOLIST**

Turns off the listing process (including the %NOLIST line itself).

**%EJECT**

Causes a new listing page to be started.

**Note**

---

These directives have an effect only when the `-Mlist` option is used. All of the directives must begin in column one.



## OpenMP Directives

OpenMP directives in a Fortran program provide information that allows the PGF77 and PGF95 compilers to generate executable programs that use multiple threads and processors on a shared-memory parallel (SMP) computer system. An OpenMP directive may have any of the following forms:

```
!$OMP directive
C$OMP directive
*$OMP directive
```

For a complete list and specifications of OpenMP directives supported by the PGF77 and PGF95 compilers, along with descriptions of the related OpenMP runtime library routines, refer to [Chapter 8, “OpenMP Directives for Fortran,”](#) on page 267.

## HPF Directives

HPF directives in a Fortran program provide information that allows the PGHPF compiler to explicitly create data distributions from which parallelism can be derived. An HPF directive may have any of the following forms:

```
CHPF$ directive
!HPF$ directive
*HPF$ directive
```

Since HPF supports two source forms, fixed source form and free source form, there are a variety of methods to enter a directive.

- For fixed source form directive, the C, !, or \* must be in column 1.
- In free source form, Fortran limits the comment character to !.

If you use the !HPF\$ form for the directive origin, your code will be universally valid.

- The body of the directive may immediately follow the directive prefix or may follow any number of blanks.
- Any names in the body of the directive, including the directive name, may not contain embedded blanks.
- Blanks may surround any special characters, such as a comma or an equals sign.
- The directive name, including the directive origin, may contain upper or lower case letters (case is not significant).

For more information on a complete list and specifications of HPF directives supported by the PGHPF compiler, refer to [Chapter 10, “HPF,”](#) on page 311.



# Chapter 2. Fortran Data Types

Every Fortran element and expression has a data type. The data type of an element may be implicit in its definition or explicitly attached to the element in a declaration statement. This chapter describes the Fortran data types and constants that are supported by the PGI Fortran compilers.

Fortran provides two kinds of data types, intrinsic data types and derived data types. Types provided by the language are intrinsic types. Types specified by the programmer and built from the intrinsic data types are called derived types.

## Intrinsic Data Types

Fortran provides six different intrinsic data types, listed in [Table 2.1, “Fortran Intrinsic Data Types”](#) and [Table 2.3, “Data Type Extensions”](#) show variations and different *KIND* of intrinsic data types supported by the PGI Fortran compilers.

Table 2.1. Fortran Intrinsic Data Types

Data Type	Value
INTEGER	An integer number.
REAL	A real number.
DOUBLE PRECISION	A double precision floating point number, real number, taking up two numeric storage units and whose precision is greater than REAL.
LOGICAL	A value which can be either TRUE or FALSE.
COMPLEX	A pair of real numbers used in complex arithmetic. Fortran provides two precisions for COMPLEX numbers.
CHARACTER	A string consisting of one or more printable characters.

## Kind Parameter

The Fortran 95 *KIND* parameter specifies a precision for intrinsic data types. The *KIND* parameter follows a data type specifier and specifies size or type of the supported data type. A *KIND* specification overrides the length attribute that the statement implies and assigns a specific length to the item, regardless of the compiler's

command-line options. A KIND is defined for a data type by a PARAMETER statement, using sizes supported on the particular system.

The following are some examples using a KIND specification:

```
INTEGER (SHORT) :: L
REAL (HIGH) B
REAL (KIND=HIGH) XVAR, YVAR
```

These examples require that the programmer use a PARAMETER statement to define kinds:

```
INTEGER, PARAMETER :: SHORT=1
INTEGER HIGH
PARAMETER (HIGH=8)
```

The following table shows several examples of KINDs that a system could support.

Table 2.2. Data Types Kind Parameters

Type	Kind	Size
INTEGER	SHORT	1 byte
INTEGER	LONG	4 bytes
REAL	HIGH	8 bytes

### Number of Bytes Specification

The PGI Fortran compilers support a length specifier for some data types. The data type can be followed by a data type length specifier of the form \*s, where s is one of the supported lengths for the data type. Such a specification overrides the length attribute that the statement implies and assigns a specific length to the specified item, regardless of the compiler options. For example, REAL\*8 is equivalent to DOUBLE PRECISION.

Table 2.3 shows the lengths of data types, their meanings, and their sizes.

Table 2.3. Data Type Extensions

Type	Meaning	Size
LOGICAL*1	Small LOGICAL	1 byte
LOGICAL*2	Short LOGICAL	2 bytes
LOGICAL*4	LOGICAL	4 bytes
LOGICAL*8	LOGICAL	8 bytes
BYTE	Small INTEGER	1 byte
INTEGER*1	Same as BYTE	1 byte
INTEGER*2	Short INTEGER	2 bytes
INTEGER*4	INTEGER	4 bytes
INTEGER*8	INTEGER	8 bytes
REAL*4	REAL	4 bytes
REAL*8	DOUBLE PRECISION	8 bytes

Type	Meaning	Size
COMPLEX*8 COMPLEX (Kind=4)	COMPLEX	8 bytes
COMPLEX*16 COMPLEX (Kind=8)	DOUBLE COMPLEX	16 bytes

The BYTE type is treated as a signed one-byte integer and is equivalent to LOGICAL\*1.

### Note

---

Assigning a value too big for the data type to which it is assigned is an undefined operation.

A specifier is allowed after a CHARACTER function name even if the CHARACTER type word has a specifier. In the following example, the function size specification C\*8 overrides the CHARACTER\*4 specification.

```
CHARACTER*4 FUNCTION C*8 (VAR1)
```

Logical data items can be used with any operation where a similar sized integer data item is permissible and vice versa. The logical data item is treated as an integer or the integer data item is treated as a logical of the same size and no type conversion is performed.

Floating point data items of type REAL or DOUBLE PRECISION may be used as array subscripts, in computed GOTOS, in array bounds and in alternate returns. The floating point data item is converted to an integer.

The data type of the result of an arithmetic expression corresponds to the type of its data. The type of an expression is determined by the rank of its elements. [Table 2.4](#) shows the ranks of the various data types, from lowest to highest.

Table 2.4. Data Type Ranks

Data Type	Rank
LOGICAL	1 (lowest)
LOGICAL*8	2
INTEGER*2	3
INTEGER*4	4
INTEGER*8	5
REAL*4	6
REAL*8 (Double precision)	7
COMPLEX*8 (Complex)	8
COMPLEX*16 (Double complex)	9 (highest)

The data type of a value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. The exception to this rule is that an operation

involving a COMPLEX\*8 element and a REAL\*8 element produces a COMPLEX\*16 result. In this operation, the COMPLEX\*8 element is converted to a COMPLEX\*16 element, which consists of two REAL\*8 elements, before the operation is performed.

In most cases, a logical expression will have a LOGICAL\*4 result. In cases where the hardware supports LOGICAL\*8 and if the expression is LOGICAL\*8, the result may be LOGICAL\*8.

## Constants

A constant is an unchanging value that can be determined at compile time. It takes a form corresponding to one of the data types. The PGI Fortran compilers support decimal (INTEGER and REAL), unsigned binary, octal, hexadecimal, character and Hollerith constants.

The use of character constants in a numeric context, for example, in the right-hand side of an arithmetic assignment statement, is supported. These constants assume a data type that conforms to the context in which they appear.

### Integer Constants

The form of a decimal integer constant is:

```
[s]d1d2...dn [ _ kind-parameter ]
```

where *s* is an optional sign and *d<sub>i</sub>* is a digit in the range 0 to 9. The optional *\_kind#parameter* is a Fortran 90/95 feature supported only by PGF95 and PGHPE, and specifies a supported kind. The value of an integer constant must be within the range for the specified kind.

The value of an integer constant must be within the range  $-2^{31}$  to  $2^{31} - 1$  inclusive. Integer constants assume a data type of INTEGER\*4 and have a 32-bit storage requirement.

The `-i8` compilation option causes all data of type INTEGER to be promoted to an 8 byte INTEGER. The `-i8` option does not override an explicit data type extension size specifier, such as INTEGER\*4. The range, data type and storage requirement change if the `-i8` flag is specified, although this flag is not supported on all x86 targets. With the `-i8` flag, the range for integer constants is  $-2^{63}$  to  $(2^{63} - 1)$ , and in this case the value of an integer constant must be within the range -9223372036854775808 to 9223372036854775807. If the constant does not fit in an INTEGER\*4 range, the data type is INTEGER\*8 and the storage requirement is 64 bits.

Here are several examples of integer constants:

```
+2
-36
437
-36_SHORT
369_I2
```

### Binary, Octal and Hexadecimal Constants

The PGI compilers and Fortran 90/95 support various types of constants in addition to decimal constants. Fortran allows unsigned binary, octal, or hexadecimal constants in DATA statements. PGI compilers support these constants in DATA statements, and additionally, support some of these constants outside of DATA statements. For more information on support of these constants, refer to [“Fortran Binary, Octal and Hexadecimal Constants,”](#) on page 25.

## Real Constants

Real constants have two forms, scaled and unscaled. An unscaled real constant consists of a signed or unsigned decimal number (a number with a decimal point). A scaled real constant takes the same form as an unscaled constant, but is followed by an exponent scaling factor of the form:

```
E+digits [ _ kind-parameter ]
Edigit [ _ kind-parameter ]
E-digits [ _ kind-parameter ]
```

where digits is the scaling factor, the power of ten, to be applied to the unscaled constant. The first two forms above are equivalent, that is, a scaling factor without a sign is assumed to be positive. [Table 2.5](#) shows several real constants.

Table 2.5. Examples of Real Constants

Constant	Value
1.0	unscaled single precision constant
1.	unscaled single precision constant
-.003	signed unscaled single precision constant
-.003_LOW	signed unscaled constant with kind LOW
-1.0	signed unscaled single precision constant
6.1E2_LOW	is equivalent to 610.0 with kind LOW
+2.3E3_HIGH	is equivalent to 2300.0 with kind HIGH
6.1E2	is equivalent to 610.0
+2.3E3	is equivalent to 2300.0
-3.5E-1	is equivalent to -0.35

## Double Precision Constants

A double precision constant has the same form as a scaled REAL constant except that the E is replaced by D and the kind parameter is not permitted. For example:

```
D+digits
Ddigit
D-digits
```

[Table 2.6](#) shows several double precision constants.

Table 2.6. Examples of Double Precision Constants

Constant	Value
6.1D2	is equivalent to 610.0
+2.3D3	is equivalent to 2300.0

Constant	Value
-3.5D-1	is equivalent to -0.35
+4D4	is equivalent to 40000

## Complex Constants

A complex constant is held as two real or integer constants separated by a comma and surrounded by parentheses. The first real number is the real part and the second real number is the imaginary part. Together these values represent a complex number. Integer values supplied as parameters for a COMPLEX constant are converted to REAL numbers. Here are several examples:

```
(18, -4)
(3.5, -3.5)
(6.1E2, +2.3E3)
```

## Double Complex Constants

A complex constant is held as two double constants separated by a comma and surrounded by parentheses. The first double is the real part and the second double is the imaginary part. Together these values represent a complex number. Here is an example:

```
(6.1D2, +2.3D3)
```

## Logical Constants

A logical constant is one of:

```
.TRUE. [ _ kind-parameter ]
.FALSE. [ _ kind-parameter ]
```

The logical constants `.TRUE.` and `.FALSE.` are by default defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be `.TRUE.` if its least significant bit is 1 and `.FALSE.` otherwise<sup>1</sup>.

Here are several examples:

```
.TRUE.
.FALSE.
.TRUE._BIT
```

The abbreviations `.T.` and `.F.` can be used in place of `.TRUE.` and `.FALSE.` in data initialization statements and in NAMELIST input.

## Character Constants

Character string constants may be delimited using either an apostrophe (') or a double quote ("). The apostrophe or double quote acts as a delimiter and is not part of the character constant. Use double quotes or two apostrophes together to include an apostrophe as part of an expression. If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escape. Within character constants, blanks are significant. For further information on the use of the backslash character, refer to `-Mbackslash` in the PGI User's Guide.

<sup>1</sup>The option `-Munixlogical` defines a logical expression to be TRUE if its value is non-zero, and FALSE otherwise; also, the internal value of `.TRUE.` is set to one. This option is not available on all target systems.



A character constant is one of:

```
[ kind-parameter_ ] "[characters]"
[ kind-parameter_ ] '[characters]'
```

Here are several examples of character constants.

```
'abc'
'abc '
'ab' 'c'
"Test Word"
GREEK_"μ"
```

A zero length character constant is written as '' or ''''.

If a character constant is used in a numeric context, for example as the expression on the right side of an arithmetic assignment statement, it is treated as a Hollerith constant. The rules for typing and sizing character constants used in a numeric context are described in [“Hollerith Constants,” on page 27](#).

## PARAMETER Constants

The PARAMETER statement permits named constants to be defined. For more details on defining constants, refer to the description of the PARAMETER statement in [Chapter 3, “Fortran Statements”](#).

## Derived Types

A derived type is a type made up of components whose type is either intrinsic or another derived type. The TYPE and END TYPE keywords define a derived type. For example, the following derived type declaration defines the type PERSON and the array CUSTOMER of type PERSON:

```
! Declare a structure to define a person derived type
TYPE PERSON
  INTEGER ID
  LOGICAL LIVING
  CHARACTER(LEN=20) FIRST, LAST, MIDDLE
  INTEGER AGE
END TYPE PERSON
TYPE (PERSON) CUSTOMER(10)
```

A derived type statement definition is called a derived-type statement (the statements between TYPE PERSON and END TYPE PERSON in the previous example. The definition of a variable of the new type is called a TYPE statement (CUSTOMER in the previous example); note the use of parentheses in the TYPE statement.

The % character accesses the components of a derived type. For example:

```
CUSTOMER(1)%ID = 11308
```

## Arrays

Arrays in Fortran are not data types, but are data objects of intrinsic or derived type with special characteristics. A dimension statement provides a data type with one or more dimensions. There are several differences between Fortran 90/95 and traditional FORTRAN 77 arrays.

### Note

---

Fortran 90/95 supports all FORTRAN 77 array semantics.

An array is a group of consecutive, contiguous storage locations associated with a symbolic name which is the array name. Each individual element of storage, called the array element, is referenced by the array name modified by a list of subscripts. Arrays are declared with type declaration statements, DIMENSION statements and COMMON statements; they are not defined by implicit reference. These declarations will introduce an array name and establish the number of dimensions and the bounds and size of each dimension. If a symbol, modified by a list of subscripts is not defined as an array, then it will be assumed to be a FUNCTION reference with an argument list.

Fortran 90/95 arrays are “objects” and operations and expressions involving arrays may apply to every element of the array in an unspecified order. For example, in the following code, where A and B are arrays of the same shape (conformable arrays), the following expression adds six to every element of B and assigns the results to the corresponding elements of A:

```
A = B + 6
```

Fortran arrays may be passed with unspecified shapes to subroutines and functions, and sections of arrays may be used and passed as well. Arrays of derived type are also valid. In addition, allocatable arrays may be created with deferred shapes (number of dimensions is specified at declaration, but the actual bounds and size of each dimension are determined when the array is allocated while the program is running).

## Array Declaration Element

An array declaration has the following form:

```
name( [lb: ]ub[ , [lb: ]ub] . . . )
```

where name is the symbolic name of the array, lb is the specification of the lower bound of the dimension and ub is the specification of the upper bound. The upper bound, ub must be greater than or equal to the lower bound lb. The values lb and ub may be negative. The bound lb is taken to be 1 if it is not specified. The difference (ub-lb+1) specifies the number of elements in that dimension. The number of lb, ub pairs specifies the rank of the array. Assuming the array is of a data type that requires N bytes per element, the total amount of storage of the array is:

```
N * (ub-lb+1) * (ub-lb+1) * . . .
```

The dimension specifiers of an array subroutine argument may themselves be subroutine arguments or members of COMMON.

## Deferred Shape Arrays

Deferred-shape arrays are those arrays whose shape can be changed by an executable statement. Deferred-shape arrays are declared with a rank, but with no bounds information. They assume their shape when either an ALLOCATE statement or a REDIMENSION statement is encountered.

For example, the following statement declares a deferred shape REAL array A of rank two:

```
REAL A( : , : )
```

## Subscripts

A subscript is used to specify an array element for access. An array name qualified by a subscript list has the following form:

```
name(sub[,sub]...)
```

where there must be one sub entry for each dimension in array name.

Each sub must be an integer expression yielding a value which is within the range of the lower and upper bounds. Arrays are stored as a linear sequence of values in memory and are held such that the first element is in the first store location and the last element is in the last store location. In a multi-dimensional array the first subscript varies more rapidly than the second, the second more rapidly than the third, and so on (column major order).

## Character Substring

A character substring is a contiguous portion of a character variable and is of type character. A character substring can be referenced, assigned values and named. It can take either of the following forms:

```
character_variable_name(x1:x2)
character_array_name(subscripts)(x1:x2)
```

where  $x1$  and  $x2$  are integers and  $x1$  denotes the left-hand character position and  $x2$  the right-hand one. These are known as substring expressions. In substring expressions  $x1$  must be both greater than or equal to 1 and less than  $x2$  and  $x2$  must be less than or equal to the length of the character variable or array element.

For example, the following expression indicates the characters in positions 2 to 4 of character variable  $J$ .

```
J(2:4)
```

This next expression indicates characters in positions 1 to 4 of array element  $K(3,5)$ .

```
K(3,5)(1:4)
```

A substring expression can be any valid integer expression and may contain array elements or function references.

## Fortran Pointers and Targets

Fortran pointers are similar to allocatable arrays. Pointers are declared with a type and a rank; they do not actually represent a value, however, but represent a value's address. Fortran 90/95 has a specific assignment operator, `=>`, for use in pointer assignments.

## Fortran Binary, Octal and Hexadecimal Constants

The PGI Fortran compilers support two representations for binary, octal, and hexadecimal numbers: the standard Fortran 90/95 representation and the PGI extension representation. In addition, PGI supports an alternate representation, described in the next section.

Fortran supports binary, octal and hexadecimal constants in DATA statements.

### Binary Constants

The form of a binary constant is:

```
B'b1b2...bn'
B"b1b2...bn"
```

where  $b_i$  is either 0 or 1., such as B'01001001'

## Octal Constants

The form of an octal constant is:

```
O'c1c2...cn'  
O"c1c2...cn"
```

where  $c_i$  is in the range 0 through 7, such as `O'043672'`

## Hexidecimal Constants

The form of a hexadecimal constant is:

```
Z'a1a2...an'  
Z"a1a2...an"
```

or

```
'a1a2...an'X  
"a1a2...an"X
```

where  $a_i$  is in the range 0 through 9 or a letter in the range A through F or a through f (case mixing is allowed), such as `Z'8473Abc58'` or `"BF40289cd"X`.

## Octal and Hexadecimal Constants - Alternate Forms

The PGF95 and PGHPF compilers support additional extensions. This is an alternate form for octal constants, outside of DATA statements. The form for an octal constant is:

```
'c1c2...cn'O
```

The form of a hexadecimal constant is:

```
'a1a2...an'X
```

where  $c_i$  is a digit in the range 0 to 7 and  $a_i$  is a digit in the range 0 to 9 or a letter in the range A to F or a through f (case mixing is allowed). Up to 64 bits (22 octal digits or 16 hexadecimal digits) can be specified.

Octal and hexadecimal constants are stored as either 32-bit or 64-bit quantities. They are padded on the left with zeroes if needed and assume data types based on how they are used.

The following are the rules for converting these data types:

- A constant is always either 32 or 64 bits in size and is typeless. Sign-extension and type-conversion are never performed. All binary operations are performed on 32-bit or 64-bit quantities. This implies that the rules to follow are only concerned with mixing 32-bit and 64-bit data.
- When a constant is used with an arithmetic binary operator (including the assignment operator) and the other operand is typed, the constant assumes the type and size of the other operand.
- When a constant is used in a relational expression such as `.EQ.`, its size is chosen from the operand having the largest size. This implies that 64-bit comparisons are possible.
- When a constant is used as an argument to the generic AND, OR, EQV, NEQV, SHIFT, or COMPL function, a 32-bit operation is performed if no argument is more than 32 bits in size; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument in any other context, no data type is assumed; however, a length of four bytes is always used. If necessary, truncation on the left occurs.

- When a specific 32-bit or 64-bit data type is required, that type is assumed for the constant. Array subscripting is an example.
- When a constant is used in a context other than those mentioned above, an INTEGER\*4 data type is assumed. Logical expressions and binary arithmetic operations with other untyped constants are examples.
- When the required data type for a constant implies that the length needed is more than the number of digits specified, the leftmost digits have a value of zero. When the required data type for a constant implies that the length needed is less than the number of digits specified, the constant is truncated on the left. Truncation of nonzero digits is allowed.

In the example below, the constant I (of type INTEGER\*4) and the constant J (of type INTEGER\*2) will have hex values 1234 and 4567, respectively. The variable D (of type REAL\*8) will have the hex value x4000012345678954 after its second assignment:

```
I = '1234'X ! Leftmost Pad with zero
J = '1234567'X ! Truncate Leftmost 3 hex digits
D = '40000123456789ab'X
D = NEQV(D,'ff'X) ! 64-bit Exclusive Or
```

## Hollerith Constants

The form of a Hollerith constant is:

```
nHc1c2...cn
```

where *n* specifies the positive number of characters in the constant and cannot exceed 2000 characters.

A Hollerith constant is stored as a byte string with four characters per 32-bit word. Hollerith constants are untyped arrays of INTEGER\*4. The last word of the array is padded on the right with blanks if necessary. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

The data type of a Hollerith constant used in a numeric expression is determined by the following rules:

- Sign-extension is never performed.
- The byte size of the Hollerith constant is determined by its context and is not strictly limited to 32 or 64 bits like hexadecimal and octal constants.
- When the constant is used with a binary operator (including the assignment operator), the data type of the constant assumes the data type of the other operand.
- When a specific data type is required, that type is assumed for the constant. When an integer or logical is required, INTEGER\*4 and LOGICAL\*4 are assumed. When a float is required, REAL\*4 is assumed (array subscripting is an example of the use of a required data type).
- When a constant is used as an argument to certain generic functions (AND, OR, EQV, NEQV, SHIFT, and COMPL), a 32-bit operation is performed if no argument is larger than 32 bits; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument, no data type is assumed and the argument is passed as an INTEGER\*4 array. Character constants are passed by descriptor only.
- When a constant is used in any other context, a 32-bit INTEGER\*4 array type is assumed.

When the length of the Hollerith constant is less than the length implied by the data type, spaces 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.

## Structures

A structure, a DEC extension to FORTRAN 77, is a user-defined aggregate data type having the following form:

```
STRUCTURE [/structure_name/][field_namelist]
  field_declaration
  [field_declaration]
  ...
  [field_declaration]
END STRUCTURE
```

Where:

*structure\_name*

is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

*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*

can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

The following rules apply:

- Field names within the same declaration nesting level must be unique.
- An inner structure declaration can include field names used in an outer structure declaration without conflict.
- Records use periods to separate fields, so it is not legal to use relational operators (for example, `.EQ.`, `.XOR.`), logical constants (`.TRUE.` or `.FALSE.`), or logical expressions (`.AND.`, `.NOT.`, `.OR.`) as field names in structure declarations.
- Fields within structures conform to machine-dependent alignment requirements, that is, fields in a structure are aligned as required by hardware.
  - A structure's storage requirements are machine-dependent.
  - Alignment of fields provides a C-like "struct" building capability and allows convenient inter-language communications.
- Because explicit padding of records is not necessary, the compiler recognizes the `%FILL` intrinsic, but performs no action in response to it.
- Data initialization can occur for the individual fields.

## Records

A record, a DEC extension to FORTRAN 77, is a user-defined aggregate data item having the following form:

```

RECORD /structure_name/record_namelist
  [ ,/structure_name/record_namelist]
  ...
  [ ,/structure_name/record_namelist]

```

where:

**structure\_name**

is the name of a previously declared structure.

**record\_namelist**

is a list of one or more variable or array names separated by commas.

You create memory storage for a record by specifying a structure name in the **RECORD** statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, recordname.fieldname). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, **INTEGER**), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of **COMMON**, **SAVE**, **NAMELIST**, **DATA** and **EQUIVALENCE** statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

The following example shows **RECORD** and **STRUCTURE** usage.

```

STRUCTURE /person/
! Declare a structure defining a person
! Person has id, names, age, and may or not be living
  INTEGER id
  LOGICAL living
  CHARACTER*5 first, last, middle
  INTEGER age
END STRUCTURE

! Define population to be an array where each element is of
! type person. Also define a variable, me, of type person.
RECORD /person/ population(2), me
...
me.age = 34           ! Assign values for the variable me
me.living = .TRUE.   ! to some of the fields.
me.first = 'Steve'
me.id = 542124822
...
population(1).last = 'Jones' ! Assign the "last" field of
                             ! element 1 of array population.
population(2) = me         ! Assign all values of record
                             ! "me" to the record population(2)
...

```

## UNION and MAP Declarations

A **UNION** declaration, a DEC extension to **FORTRAN 77**, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure.

## Declaring and Defining Fields

Each group of fields within a UNION declaration is declared by a MAP declaration, with one or more fields per MAP declaration.

You use union declarations when you want to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration.

## Format

The format of a UNION statement is illustrated in the following example:

```
UNION
  map_declaration
  [map_declaration]
  ...
  [map_declaration]
END UNION
```

The format of the map\_declaration is as follows:

```
MAP
  field_declaration
  [field_declaration]
  ...
  [field_declaration]
END MAP
```

where *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 typed data field within a union.

With respect to UNION and MAP statements, the following is true:

- Data can be initialized in field declaration statements in union declarations.

### Note

---

It is illegal to initialize multiple map declarations in a single union.

- Field alignment within multiple map declarations is performed as previously defined in structure declarations.
- The size of the shared area 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 field(s) declared within it plus the space reserved for alignment purposes.



Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the C language requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following example shows RECORD, STRUCTURE, MAP and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP, in this case, the employee map (24 bytes).

```

STRUCTURE /account/
  INTEGER typetag      ! Tag to determine defined map.
  UNION
  MAP                  ! Structure for an employee
  CHARACTER*12 ssn    ! Social Security Number
  REAL*4 salary       ! Salary
  CHARACTER*8 empdate ! Employment date
  END MAP

  MAP                  ! Structure for a customer
  INTEGER*4 acct_cust ! 4-digit account
  REAL*4 credit_amt   ! credit amount
  CHARACTER*8 due_date ! due date
  END MAP

  MAP                  ! Structure for a supplier
  INTEGER*4 acct_supp ! supply account
  REAL*4 debit_amt    ! debit amount
  BYTE num_items      ! number of items
  BYTE items(12)      ! items supplied
  END MAP

  END UNION
END STRUCTURE
RECORD /account/ recarr(1000)

```

## Data Initialization

Data initialization is allowed within data type declaration statements. This is an extension to the Fortran language. Data is initialized by placing values bounded by slashes immediately following the symbolic name (variable or array) to be initialized. Initialization of fields within structure declarations is allowed, but initialization of unnamed fields and records is not.

Hollerith, octal and hexadecimal constants can be used to initialize data in both data type declarations and in DATA statements. Truncation and padding occur for constants that differ in size from the declared data item (as specified in the discussion of constants).

## Pointer Variables

The POINTER statement, a CRAY extension to FORTRAN 77 which is distinct from the Fortran 90/95 POINTER specification statement or attribute, declares a scalar variable to be a pointer variable of data type INTEGER, and another variable to be its pointer-based variable.

The syntax of the POINTER statement is:

```
POINTER (p1, v1) [, (p2, v2) ...]
```

**v1 and v2**

are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.

**p1 and p2**

are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined.

The pointer is defined in one of the following ways:

- By assigning the value of the LOC function.
- By assigning a value defined in terms of another pointer variable.
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

The following code illustrates the use of pointers:

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0 ! IC gets 0
P = LOC(XC)
Q = P + 20 ! same as LOC(XC(6))
X(1) = 0 ! XC(6) gets 0
ALLOCATE (X) ! Q locates an allocated memory area
```

## Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram.

If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarators are defined with an integer value at the time the SUBROUTINE or FUNCTION is called.

For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.

- A pointer-based variable is assumed not to overlap with another pointer-based variable.

## Pointer Assignment

Fortran 2003 extends pointer assignment for arrays allowing lower bounds and possibly upper bounds to be specified.

Syntax:

```
p(0:,0:) => a
```

The lower bounds may be any scalar integer expressions when upper bounds are specified. Further, remapping of the elements of a target array is permitted, as shown in this example:

```
p(1:m,1:2*m) => a(1:2*m)
```

## Description

The following is true for pointer assignments involving arrays:

- The bounds may be any scalar integer expressions.
- The assignment is in array-element order and the target array must be large enough.
- When remapping occurs, the target must be rank-one; otherwise, the ranks of the pointer and target must be the same.

```
a => b(1:10:2)
```

- Length type parameters of the pointer may be deferred, that is, declared with a colon.
- Pointer assignment gives these the values of the corresponding parameters of the target.
- All other type parameters of the pointer must have the same values as the corresponding type parameters of the target.



# Chapter 3. Fortran Statements

This chapter describes each of the Fortran statements supported by the PGI Fortran compilers. Each description includes a brief summary of the statement, a syntax description, a complete description and an example. The statements are listed in alphabetical order. The first section lists terms that are used throughout the chapter.

## Statement Format Overview

The following sections contain detailed descriptions of each of the statements. This section lists terms that are used throughout the chapter and provides information on how to interpret the information in these statement descriptions.

### Definition of Statement-related Terms

character scalar memory reference

is a character variable, a character array element, or a character member of a structure or derived type.

integer scalar memory reference

is an integer variable, an integer array element, or an integer member of a structure or derived type.

logical scalar memory reference

is a logical variable, a logical array element, or a logical member of a structure or derived type.

obsolescent

The statement is unchanged from the FORTRAN 77 definition but has a better replacement in Fortran 95.

### Origin of Statement

At the top of each reference page is a brief description of the statement followed by a header that indicates the origin of the statement. The following list describes the meaning of the origin header.

F77

FORTRAN 77 statements that are essentially unchanged from the original FORTRAN 77 standard and are supported by the PGF77 compiler.

F77 extension

The statement is an extension to the Fortran language.

**F90/F95**

This statement is either new for Fortran 90/95 or significantly changed in Fortran 95 from its original FORTRAN 77 definition and is supported by the PGF95 and PGHPF compilers.

**HPF (High Performance Fortran)**

The statement has its origin in the HPF standard.

**CMF**

Indicates a CM Fortran feature (CM Fortran is a version of Fortran that was produced by Thinking Machines Corporation for parallel computers).

**List-related Notation**

Several statements allow lists of a specific type of data. For example, the ALLOCATABLE statement allows a list in which each element of a deferred-array-spec. The notation used in statements is this:

- Within the statement, the notation is `foo-list`, such as `deferred-array-spec-list`.
- When the list elements have a specific format that is defined, the reference is just to that element, such as `deferred-array-spec`.

As in Fortran, the list is a set of comma-separated values.

**Fortran Statement Summary Table**

This section contains an alphabetical listing with a brief one-line description of the Fortran statements included in this chapter.

Table 3.1. Intrinsic Summary Table

<b>Intrinsic</b>		<b>Description</b>
<b>ACCEPT</b>	F77	Causes formatted input to be read on standard input.
<b>ALLOCATABLE</b>	F90	Specifies that an array with fixed rank but deferred shape is available for a future ALLOCATE statement.
<b>ALLOCATE</b>	F90	Allocates storage for each allocatable array, pointer object, or pointer-based variable that appears in the statements; declares storage for deferred-shape arrays.
<b>ARRAY</b>	CMF	Defines the number of dimensions in an array that may be defined, and the number of elements and bounds in each dimension.
<b>ASSIGN</b>	F77	[Obsolescent]. Assigns a statement label to a variable.
<b>BACKSPACE</b>	F77	Positions the file connected to the specified unit to before the preceding record.
<b>BLOCK DATA</b>	F77	Introduces a number of non-executable statements that initialize data values in COMMON tables
<b>BYTE</b>	F77 ext	Establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer, overriding implied data typing.
<b>CALL</b>	F77	Transfers control to a subroutine.

Intrinsic		Description
<b>CASE</b>	F90	Begins a case-statement-block portion of a SELECT CASE statement.
<b>CHARACTER</b>	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a character data type, overriding the implied data typing.
<b>CLOSE</b>	F77	Terminates the connection of the specified file to a unit.
<b>COMMON</b>	F77	Defines global blocks of storage that are either sequential or non-sequential; can be either a static or dynamic form.
<b>COMPLEX</b>	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type, overriding implied data typing.
<b>CONTAINS</b>	F90	Precedes a subprogram, a function or subroutine and indicates the presence of the subroutine or function definition inside a main program, external subprogram, or module subprogram.
<b>CONTINUE</b>	F77	Passes control to the next statement.
<b>CYCLE</b>	F90	Interrupts a DO construct execution and continues with the next iteration of the loop.
<b>DATA</b>	F77	Assigns initial values to variables before execution.
<b>DEALLOCATE</b>	F77	Causes the memory allocated for each pointer-based variable or allocatable array that appears in the statement to be deallocated (freed); also deallocates storage for deferred-shape arrays.
<b>DECODE</b>	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers.
<b>DIMENSION</b>	F90	Defines the number of dimensions in an array and the number of elements in each dimension.
<b>DO (Iterative)</b>	F90	Introduces an iterative loop and specifies the loop control index and parameters.
<b>DO WHILE</b>	F77	Introduces a logical do loop and specifies the loop control expression.
<b>DOUBLE COMPLEX</b>	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type, overriding implied data typing.
<b>DOUBLE PRECISION</b>	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type, overriding implied data typing.
<b>ELSE</b>	F77	Begins an ELSE block of an IF block and encloses a series of statements that are conditionally executed.
<b>ELSE IF</b>	F77	Begins an ELSE IF block of an IF block series and encloses statements that are conditionally executed.

Intrinsic		Description
<a href="#">ELSE WHERE</a>	F77	The portion of the WHERE ELSE WHERE construct that permits conditional masked assignments to the elements of an array or to a scalar, zero-dimensional array.
<a href="#">ENCODE</a>	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers.
<a href="#">END</a>	F77	Terminates a segment of a Fortran program.
<a href="#">END DO</a>	F77	Terminates a DO or DO WHILE loop.
<a href="#">END FILE</a>	F77	Writes an endfile record to the files.
<a href="#">END IF</a>	F77	Terminates an IF ELSE or ELSE IF block.
<a href="#">END MAP</a>	F77 ext	Terminates a MAP declaration.
<a href="#">END SELECT</a>	F77	Terminates a SELECT declaration.
<a href="#">END STRUCTURE</a>	F77 ext	Terminates a STRUCTURE declaration.
<a href="#">END UNION</a>	F77 ext	Terminates a UNION declaration.
<a href="#">END WHERE</a>	F77 ext	Terminates a WHERE ELSE WHERE construct.
<a href="#">ENTRY</a>	F77	Allows a subroutine or function to have more than one entry point.
<a href="#">EQUIVALENCE</a>	F77	Allows two or more named regions of data memory to share the same start address.
<a href="#">EXIT</a>	F90	Interrupts a DO construct execution and continues with the next statement after the loop.
<a href="#">EXTERNAL</a>	F77	Identifies a symbolic name as an external or dummy procedure which can then be used as an actual argument.
<a href="#">EXTRINSIC</a>	HPF	Identifies a symbolic name as an external or dummy procedure that is written in some language other than HPF.
<a href="#">FORALL</a>	F95	Provides, as a statement or construct, a parallel mechanism to assign values to the elements of an array.
<a href="#">FORMAT</a>	F77	Specifies format requirements for input or output.
<a href="#">FUNCTION</a>	F77	Introduces a program unit; all the statements that follow apply to the function itself.
<a href="#">GOTO (Assigned)</a>	F77	[Obsolescent]. Transfers control so that the statement identified by the statement label is executed next.
<a href="#">GOTO (Computed)</a>	F77	Transfers control to one of a list of labels according to the value of an expression.
<a href="#">GOTO (Unconditional)</a>	F77	Unconditionally transfers control to the statement with the label <i>label</i> , which must be declared within the code of the program unit containing the GOTO statement and must be unique within that program unit.



Intrinsic		Description
IF (Arithmetic)	F77	[Obsolescent]. Transfers control to one of three labeled statements, depending on the value of the arithmetic expression.
IF (Block)	F77	Consists of a series of statements that are conditionally executed.
IF (Logical)	F77	Executes or does not execute a statement based on the value of a logical expression.
IMPLICIT	F77	Redefines the implied data type of symbolic names from their initial letter, overriding implied data types.
IMPORT	F2003	Gives access to the named entities of the containing scope.
INCLUDE	F77 ext	Directs the compiler to start reading from another file.
INQUIRE	F77	Inquires about the current properties of a particular file or the current connections of a particular unit.
INTEGER	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type, overriding implied data types.
INTENT	F90	Specifies intended use of a dummy argument, but may not be used in a main program's specification statement.
INTERFACE	F90	Makes an implicit procedure an explicit procedure where the dummy parameters and procedure type are known to the calling module; Also overloads a procedure name.
INTRINSIC	F77	Identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.
LOGICAL	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a logical data type, overriding implied data types.
MAP	F77 ext	Designates each unique field or group of fields within a UNION statement.
MODULE	F90	Specifies the entry point for a Fortran 90/95 module program unit. A module defines a host environment of scope of the module, and may contain subprograms that are in the same scoping unit.
NAMelist	F90	Allows the definition of namelist groups for namelist-directed I/O.
NULLIFY	F90	Disassociates a pointer from its target.
OPEN	F77	Connects an existing file to a unit, creates and connects a file to a unit, creates a file that is preconnected, or changes certain specifiers of a connection between a file and a unit.
OPTIONAL	F90	Specifies dummy arguments that may be omitted or that are optional.
OPTIONS	F77 ext	Confirms or overrides certain compiler command-line options.
PARAMETER	F77	Gives a symbolic name to a constant.
PAUSE	F77	[Obsolescent]. Stops the program's execution.
POINTER	F90	Provides a means for declaring pointers.

Intrinsic		Description
<b>POINTER (Cray)</b>	F77 ext	Declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its pointer-based variable.
<b>PRINT</b>	F77	Transfers data to the standard output device from the items specified in the output list and format specification.
<b>PRIVATE</b>	F90	Specifies entities defined in a module are not accessible outside of the module.
<b>PROGRAM</b>	F77	Specifies the entry point for the linked Fortran program.
<b>PUBLIC</b>	F90	Specifies entities defined in a module are accessible outside of the module.
<b>PURE</b>	F95	Indicates that a function or subroutine has no side effects.
<b>READ</b>	F90	Transfers data from the standard input device to the items specified in the input and format specifications.
<b>REAL</b>	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a data type, overriding implied data types.
<b>RECORD</b>	F77 ext	A VAX Fortran extension, defines a user-defined aggregate data item.
<b>RECURSIVE</b>	F90	Indicates whether a function or subroutine may call itself recursively.
<b>REDIMENSION</b>	F77 ext	Dynamically defines the bounds of a deferred-shape array.
<b>RETURN</b>	F77	Causes a return to the statement following a CALL when used in a subroutine, and returns to the relevant arithmetic expression when used in a function.
<b>REWIND</b>	F77	Positions the file at its beginning. The statement has no effect if the file is already positioned at the start or if the file is connected but does not exist.
<b>SAVE</b>	F77	Retains the definition status of an entity after a RETURN or END statement in a subroutine or function has been executed.
<b>SELECT CASE</b>	F90	Begins a CASE construct.
<b>SEQUENCE</b>	F90	A derived type qualifier that specifies the ordering of the storage associated with the derived type. This statement specifies storage for use with COMMON and EQUIVALENCE statements.
<b>STOP</b>	F77	Stops the program's execution and precludes any further execution of the program.
<b>STRUCTURE</b>	F77 Vax ext	A VAX extension to FORTRAN 77 that defines an aggregate data type.
<b>SUBROUTINE</b>	F77	Introduces a subprogram unit.
<b>TARGET</b>	F90	Specifies that a data type may be the object of a pointer variable (e.g., pointed to by a pointer variable). Types that do not have the TARGET attribute cannot be the target of a pointer variable.

Intrinsic		Description
<b>THEN</b>	F77	Part of a block IF statement, surrounds a series of statements that are conditionally executed.
<b>TYPE</b>	F90	Begins a derived type data specification or declares variables of a specified user-defined type.
<b>UNION</b>	F77	A multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields.
<b>USE</b>	F90	Gives a program unit access to the public entities or to the named entities in the specified module.
<b>VOLATILE</b>	F77 ext	Inhibits all optimizations on the variables, arrays and common blocks that it identifies.
<b>WHERE</b>	F90	Permits masked assignments to the elements of an array or to a scalar, zero dimensional array.
<b>WRITE</b>	F90	Transfers data to the standard output device from the items specified in the output list and format specification.

## ACCEPT

The ACCEPT statement has the same syntax as the PRINT statement and causes formatted input to be read on standard input. ACCEPT is identical to the READ statement with a unit specifier of asterisk (\*).

### F77 extension

#### Syntax

```
ACCEPT f [,iolist]
ACCEPT namelist
```

f

format-specifier, a \* indicates list directed input.

iolist

is a list of variables to be input.

namelist

is the name of a namelist specified with the NAMELIST statement.

### Examples

```
ACCEPT *, IA, ZA
ACCEPT 99, I, J, K
ACCEPT SUM
99 FORMAT(I2, I4, I3)
```

### Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

## ALLOCATABLE

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
accept ch
```

### See Also

[“READ”](#), [“PRINT”](#)

## ALLOCATABLE

The ALLOCATABLE specification statement (attribute) specifies that an array with fixed rank but deferred shape is available for a future ALLOCATE statement. An ALLOCATE statement allocates space for the allocatable object or scalar.

### F2003

#### Syntax for arrays

```
ALLOCATABLE [ :: ] array-name [(deferred-array-spec-list)]
[, array-name [(deferred-array-spec-list)]]...
```

array-name

is the name of the allocatable array.

deferred-array-spec

is a : character.

#### Syntax for scalar variables and components

```
ALLOCATABLE :: scalar-name
```

#### Description

The allocatable attribute allows the user to construct local data structures which do not leak memory because they get deallocated automatically.

#### Scalar Example

In the following example, reclen allows the length of CHARACTER to be specified at run time.

```
CHARACTER(:) ALLOCATABLE :: chdata
INTEGER :: unit, reclen
...
READ(unit) reclen
ALLOCATE(CHARACTER(reclen) :: chdata)
READ(unit) chdata
```

#### Array Example

```
REAL SCORE(:), NAMES(:, :)
REAL, ALLOCATABLE, DIMENSION(:, :, :) :: TEST
ALLOCATABLE SCORE, NAMES
```

```
INTEGER, ALLOCATABLE :: RECl( : , : , : )
```

## See Also

[“ALLOCATE”](#), [“DEALLOCATE”](#)

## ALLOCATE

The ALLOCATE statement is an extension to FORTRAN 77 but is part of the Fortran 90/95 standard. It allocates storage for each allocatable object, pointer object, or pointer-based variable that appears in the statement. ALLOCATE also declares storage for deferred-shape arrays.

## F90

### Syntax

```
ALLOCATE ( allocation-list [ , STAT= var ] )
```

allocation-list is:

```
allocate-object [ allocate-shape-spec-list ]
```

allocate-object is:

```
variable-name  
structure-component
```

allocate-shape-spec is:

```
[ allocate-lower-bound : ] allocate-upper-bound
```

var

is an integer variable, integer array element or an integer member of a derived type - that is, an integer scalar memory reference. This variable is assigned a value depending on the success of the ALLOCATE statement.

allocate-object

is a variable with the POINTER or ALLOCATABLE attribute, or name of an allocatable COMMON enclosed in slashes.

### Description

For a pointer-based variable, its associated pointer variable is defined with the address of the allocated memory area. If the specifier STAT= is present, successful execution of the ALLOCATE statement causes the status variable to be defined with a value of zero. If an error occurs during execution of the statement and the specifier STAT= is present, the status variable is defined to have the integer value one. If an error occurs and the specifier STAT= is not present, program execution is terminated.

A dynamic or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. Note: Allocatable COMMON blocks are an extension to FORTRAN 77 supported only by PGF77 compiler, and not by the PGF95 or PGHPF compilers.

For an ALLOCATABLE array, the array is allocated with the executable ALLOCATE statement.

## Examples

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

For a deferred shape array, the allocate must include the bounds of the array.

```
REAL, ALLOCATABLE :: A(:), B(:)
ALLOCATE (A(10), B(SIZE(A)))
REAL A(:, :)
N=3
M=1
ALLOCATE (A(1:11, M:N))
INTEGER FLAG, N
REAL, ALLOCATABLE:: B(:, :)
ALLOCATE (B(N,N), STAT=FLAG)
```

## ARRAY

The **ARRAY** attribute defines the number of dimensions in an array that may be defined and the number of elements and bounds in each dimension.

### CMF

### Syntax

```
ARRAY [::] array-name (array-spec)
[, array-name (array-spec) ] ...
```

**array-name**

is the symbolic name of an array.

**array-spec**

is a valid array specification, either explicit-shape, assumed-shape, deferred-shape, or assumed size (refer to [Chapter 4, “Fortran Arrays”](#), for details on array specifications).

### Description

**ARRAY** can be used in a subroutine as a synonym for **DIMENSION** to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays).

#### Note

These integer variables must be either arguments or declared in **COMMON**; they cannot be local. Further, in this case, the function of **ARRAY** statement is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

The typing of the array in an `ARRAY` statement is defined by the initial letter of the array name in the same way as variable names, unless overridden by an `IMPLICIT` or type declaration statement. Arrays may appear in type declaration and `COMMON` statements but the array name can appear in only one array declaration.

## Example

```
REAL, ARRAY(3:10):: ARRAY_ONE
INTEGER, ARRAY(3,-2:2):: ARRAY_TWO
```

This specifies `ARRAY_ONE` as a vector having eight elements with the lower bound of 3 and the upper bound of 10.

`ARRAY_TWO` as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

## See Also

[“ALLOCATE”](#), [“DEALLOCATE”](#)

## ASSIGN

(Obsolescent) The assign statement assigns a statement label to a variable. Internal procedures can be used in place of the assign statement. Other cases where the assign statement is used can be replaced by using character strings (for different format statements that were formally assigned labels by using an integer variable as a format specifier).

## F77

## Syntax

```
ASSIGN a TO b
```

a

is the statement label.

b

is an integer variable.

## Description

Executing an `ASSIGN` statement assigns a statement label to an integer variable. This is the only way that a variable may be defined with a statement label value.

The statement label must be:

- A statement label in the same program unit as the `ASSIGN` statement.
- The label of an executable statement or a `FORMAT` statement.

A variable must be defined with a statement label when it is referenced:

- In an assigned `GOTO` statement.

## BACKSPACE

- As a format identifier in an input/output statement and while so defined must not be referenced in any other way.

An integer variable defined with a statement label can be redefined with a different statement label, the same statement label or with an integer value.

### Example

```
ASSIGN 40 TO K
GO TO K
40 L = P + I + 56
```

## BACKSPACE

When a BACKSPACE statement is executed the file connected to the specified unit is positioned before the preceding record.

### F77

### Syntax

```
BACKSPACE unit
BACKSPACE ([UNIT=]unit [,ERR=errs] [,
IOSTAT=ios])
```

#### UNIT=unit

unit is the unit specifier.

#### ERR=s

s is an executable statement label for the statement used for processing an error condition.

#### IOSTAT=ios

ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

### Description

If there is no preceding record, the position of the file is not changed. A BACKSPACE statement cannot be executed on a file that does not exist. Do not issue a BACKSPACE statement for a file that is open for direct or append access.

### Examples

```
BACKSPACE 4
BACKSPACE ( UNIT=3 )
BACKSPACE ( 7, IOSTAT=IOCHEK, ERR=50 )
```

## BLOCK DATA

The BLOCK DATA statement introduces a number of statements that initialize data values in COMMON blocks. No executable statements are allowed in a BLOCK DATA segment.



## F77

## Syntax

```
BLOCK DATA [name]
  [specification]
END [BLOCK DATA [name]]
```

**name**

is a symbol identifying the name of the block data and must be unique among all global names (COMMON block names, program name, module names). If missing, the block data is given a default name.

## Example

```
BLOCK DATA
  COMMON /SIDE/ BASE, ANGLE, HEIGHT, WIDTH
  INTEGER SIZE
  PARAMETER (SIZE=100)
  INTEGER BASE(0:SIZE)
  REAL WIDTH(0:SIZE), ANGLE(0:SIZE)
  DATA (BASE(I), I=0, SIZE)/SIZE*-1, -1/,
+ (WIDTH(I), I=0, SIZE)/SIZE*0.0, 0.0/
END
```

## BYTE

The BYTE statement establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer. This overrides data typing implied by the initial letter of a symbolic name.

## F77 extension

## Syntax

```
BYTE name [/clist/],
...
```

**name**

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

**clist**

is a list of constants that initialize the data, as in a DATA statement.

## Description

Byte statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. BYTE declaration statements must not be labeled.

## Example

```
BYTE TB3, SEC, STORE (5,5)
```

## CALL

The CALL statement transfers control to a subroutine.

## F77

## Syntax

```
CALL subroutine [( [ actual-arg-list]...)]
```

**subroutine**

is the name of the subroutine.

**actual-arg-list**

is the actual argument being passed to the subroutine. The first argument corresponds to the first dummy argument in the SUBROUTINE statement and so on.

**actual-arg**

has the form:

```
[ keyword = ] subroutine-argument.
```

**keyword**

is a dummy argument name in the subroutine interface.

**subroutine-argument**

is an actual argument.

## Description

Actual arguments can be expressions including: constants, scalar variables, function references and arrays.

Actual arguments can also be alternate return specifiers. Alternate return specifiers are labels prefixed by asterisks (\*) or ampersands (&). The ampersand is an extension to FORTRAN 77.

Recursive calls are allowed using the `-Mrecursive` command-line option.

## Examples

```
CALL CRASH      ! no arguments
CALL BANG(1.0)  ! one argument
CALL WALLOP(V, INT) ! two arguments
CALL ALTRET(I, *10, *20)
SUBROUTINE ONE
DIMENSION ARR ( 10, 10 )
REAL WORK
INTEGER ROW, COL
PI=3.142857
CALL EXPENS(ARR,ROW,COL,WORK,SIN(PI/2)+3.4)
RETURN
END
```

This example illustrates subroutine calls using argument keywords. Notice that when the keyword is used, the order of the arguments is not significant.

```
INTERFACE
  SUBROUTINE SUB_1(A1,B1)
    REAL A1,B1
  END SUBROUTINE
END INTERFACE
X=7 ; Y=8
```

```
CALL SUB_1(B1=X,A1=Y)      ! subroutine call with argument keywords
END

SUBROUTINE SUB_1(D1,D2)
  REAL D1,D2,D3,D4
  D3 = D1/D2
  D4 = D1-D2
  PRINT *, D3, D4
END SUBROUTINE
```

## CASE

The CASE statement begins a case-statement-block portion of a SELECT CASE construct.

F90

### Syntax

```
[ case-name : ] SELECT CASE (case-expr)
[ CASE (selector) [name]
  block ] ...
[ CASE DEFAULT [case-name]
  block ]
END SELECT [case-name]
```

### Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
  TYPE=1
CASE ( 4:6 )
  TYPE=2
CASE DEFAULT
  TYPE=0
END SELECT
```

### Type

Executable

### See Also

[“SELECT CASE”](#)

## CHARACTER

The CHARACTER statement establishes the data type of a variable by explicitly attaching the name of a variable to a character data type. This overrides the data typing implied by the initial letter of a symbolic name.

F90

### Syntax

The syntax for CHARACTER has two forms, the standard Fortran form and the PGI extended form. This section describes both syntax forms.

## CLOSE

```
CHARACTER [character-selector] [,  
attribute-list ::] entity-list
```

### character-selector

the character selector specifies the length of the character string. This has one of several forms:

```
([LEN=] type-param-value)  
* character-length [,]
```

Character-selector also permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

### attribute-list

is the list of attributes for the character variable.

### entity-list

is the list of defined entities.

## Syntax Extension

```
CHARACTER [*len][,] name [dimension] [*len] [/clist/],  
...
```

### len

is a constant or \*. A \* is only valid if the corresponding name is a dummy argument.

### name

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

### clist

is a list of constants that initialize the data, as in a DATA statement.

## Description

Character type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled.

### Note

---

The data type of a symbol may be explicitly declared only once. It is established by a type declaration statement, an IMPLICIT statement, or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

## Examples

```
CHARACTER A*4, B*6, C  
CHARACTER (LEN=10):: NAME
```

A is 4 and B is 6 characters long and C is 1 character long. NAME is 10 characters long.

## CLOSE

The CLOSE statement terminates the connection of the specified file to a unit.

F77

## Syntax

```
CLOSE ([UNIT=] u [,ERR= errs ] [,DISP[OSE]= sta]
      [,IOSTAT=ios] [,STATUS= sta] )
```

u

is the external unit specifier where u is an integer.

errs

is an error specifier in the form of a statement label of an executable statement in the same program unit. If an error condition occurs, execution continues with the statement specified by errs.

ios

is an integer scalar; if this is included ios becomes defined with 0 (zero) if no error condition exists or a positive integer when there is an error condition. A value of -1 indicates an end-of-file condition with no error. A value of -2 indicates an end-of-record condition with no error when using non-advancing I/O.

sta

is a character expression, where case is insignificant, specifying the file status and the same keywords are used for the dispose status. Status can be 'KEEP' or 'DELETE' (the quotes are required). KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement; conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

## Description

A unit may be the subject of a CLOSE statement from within any program unit. If the unit specified does not exist or has no file connected to it the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

## Example

In the following example, the file on UNIT 8 is closed and deleted.

```
CLOSE (UNIT=8 , STATUS= ' DELETE ' )
```

**COMMON**

The COMMON statement defines global blocks of storage that are either sequential or non sequential. There are two forms of the COMMON statement, a static form and a dynamic form. Each common block is identified by the symbolic name defined in the COMMON block.

F77

## Syntax

```
COMMON /[name ] /nlist [, /name/nlist]...
```

**name**

is the name of each common block and is declared between the `/.../` delimiters for a named common and with no name for a blank common.

**nlist**

is a list of variable names where arrays may be defined in `DIMENSION` statements or formally declared by their inclusion in the `COMMON` block.

## Common Block Rules and Behaviors

A common block is a global entity. Any common block name (or blank common) can appear more than once in one or more `COMMON` statements in a program unit. The following is a list of rules associated with common blocks:

### Blank Common

The name of the `COMMON` block need not be supplied; without a name, the common is a `BLANK COMMON`. In this case the compiler uses a default name.

### Same Names

There can be several `COMMON` block statements of the same name in a program segment; these are effectively treated as one statement, with variables concatenated from one `COMMON` statement of the same name to the next. This is an alternative to the use of continuation lines when declaring a common block with many symbols.

Common blocks with the same name that are declared in different program units share the same storage area when combined into one executable program and they are defined using the `SEQUENCE` attribute.

### HPF

In HPF, a common block is non-sequential by default, unless there is an explicit `SEQUENCE` directive that specifies the array as sequential. Note this may require that older FORTRAN 77 programs assuming sequence association in `COMMON` statements have `SEQUENCE` statements for `COMMON` variables.

## Example

```
DIMENSION R(10)
COMMON /HOST/ A, R, Q(3), U
```

This declares a common block called `HOST`

### Note

---

The different types of declaration used for `R` (declared in a `DIMENSION` statement) and `Q` (declared in the `COMMON` statement).

The declaration of `HOST` in a `SUBROUTINE` in the same executable program, with a different shape for its elements would require that the array be declared using the `SEQUENCE` attribute.

```
SUBROUTINE DEMO
!HPF$ SEQUENCE HOST
COMMON/HOST/STORE(15)
.
.
```

```

.
RETURN
END

```

## Common Blocks and Subroutines

If the main program has the common block declaration as in the previous example, the COMMON statement in the subroutine causes STORE(1) to correspond to A, STORE(2) to correspond to R(1), STORE(3) to correspond to R(2), and so on through to STORE(15) corresponding to the variable U.

## Common Block Records and Characters

You can name records within a COMMON block. Because the storage requirements of records are machine-dependent, the size of a COMMON block containing records may vary between machines. Note that this may also affect subsequent equivalence associations to variables within COMMON blocks that contain records.

Both character and non-character data may reside in one COMMON block. Data is aligned within the COMMON block in order to conform to machine-dependent alignment requirements.

Blank COMMON is always saved. Blank COMMON may be data initialized.

## See Also

[“SEQUENCE”](#) and the [“SEQUENCE ”](#) directive.

## Syntax Extension – dynamic COMMON

A dynamic, or allocatable, COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. PGF77 supports dynamic COMMON blocks, while PGF95 and PGHPF do not.

If the ALLOCATABLE attribute is present, all named COMMON blocks appearing in the COMMON statement are marked as allocatable. Like a normal COMMON statement, the name of an allocatable COMMON block may appear in more than one COMMON statement. Note that the ALLOCATABLE attribute need not appear in every COMMON statement.

The following restrictions apply to the dynamic COMMON statement:

- Before members of an allocatable COMMON block can be referenced, the common block must have been explicitly allocated using the ALLOCATE statement.
- The data in an allocatable common block cannot be initialized.
- The memory used for an allocatable common block may be freed using the DEALLOCATE statement.
- If a SUBPROGRAM declares a COMMON block to be allocatable, all other subprograms containing COMMON statements of the same COMMON block must also declare the COMMON to be allocatable.

### Example (dynamic COMMON)

```

COMMON, ALLOCATABLE /ALL1/ A, B, /ALL2/ AA,
BB
COMMON /STAT/ D, /ALL1/ C

```

This declares the following variables:

## COMPLEX

### ALL1

is an allocatable COMMON block whose members are A, B, and C.

### ALL2

is an allocatable COMMON block whose members are AA, and BB.

### STAT

is a statically-allocated COMMON block whose only member is D.

A reference to a member of an allocatable COMMON block appears in a Fortran statement just like a member of a normal (static) COMMON block. No special syntax is required to access members of allocatable common blocks. For example, using the above declarations, the following is a valid pgf77 statement:

```
AA = B * D
```

## COMPLEX

The COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type. This overrides the data typing implied by the initial letter of a symbolic name.

### F90

#### Syntax

The syntax for COMPLEX has two forms, the standard Fortran form and the PGI extended form. This section describes both syntax forms.

```
COMPLEX [( [ KIND =] kind-value) ] [,attribute-list ::] entity-list
```

COMPLEX permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

#### attribute-list

is the list of attributes for the character variable.

#### entity-list

is the list of defined entities.

#### Syntax Extension

```
COMPLEX name [*n] [dimensions] [/clist/] [,name] [/clist/] ...
```

#### name

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement below for an explanation of array declarators).

#### clist

is a list of constants that initialize the data, as in a DATA statement.

#### Description

COMPLEX statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. COMPLEX statements must not be labeled.



The default size of a `COMPLEX` variable is 8 bytes. With the `-r8` option, the default size of a `COMPLEX` variable is 16 bytes.

### Note

---

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, `IMPLICIT` statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An `IMPLICIT` statement overrides predefined typing rules.

### Example

```
COMPLEX CURRENT
COMPLEX DIMENSION(8) :: CONV1, FLUX1
```

### See Also

[“DOUBLE COMPLEX”](#)

## CONTAINS

The `CONTAINS` statement precedes a subprogram, a function or subroutine, that is defined inside a main program, external subprogram, or module subprogram (internal subprogram). The `CONTAINS` statement is a flag indicating the presence of a subroutine or function definition. An internal subprogram defines a scope for the internal subprogram's labels and names. Scoping is defined by use and host scoping rules within scoping units. Scoping units have the following precedence for names:

- A derived-type definition.
- A procedure interface body.
- A program unit or a subprogram, excluding contained subprograms.

### F90

### Syntax

```
SUBROUTINE X
  INTEGER H, I
  ...
  CONTAINS
  SUBROUTINE Y
    INTEGER I
    I = I + H
    ...
  END SUBROUTINE Y
END SUBROUTINE X
```

### Type

Non-executable

### See Also

[“MODULE”](#)

## CONTINUE

The CONTINUE statement passes control to the next statement. It is supplied mainly to overcome the problem that transfer of control statements are not allowed to terminate a DO loop.

F77

### Syntax

```
CONTINUE
```

### Example

```
DO 100 I = 1,10
SUM = SUM + ARRAY (I)
IF(SUM .GE. 1000.0) GOTO 200
100 CONTINUE
200 ...
```

### See Also

[“GOTO \(Computed\)”](#), [“GOTO \(Unconditional\)”](#)

## CYCLE

The CYCLE statement interrupts a DO construct execution and continues with the next iteration of the loop.

F90

### Syntax

```
CYCLE [do-construct-name]
```

### Example

```
DO
IF (A(I).EQ.0) CYCLE
B=100/A(I)
IF (B.EQ.5) EXIT
END DO
```

### See Also

[“EXIT”](#), [“DO \(Iterative\)”](#)

## DATA

The DATA statement assigns initial values to variables before execution.

F77

### Syntax

```
DATA vlist/dlist/[[, ]vlist/dlist/]...
```

vlist

is a list of variable names, array element names or array names separated by commas.

dlist

is a list of constants or PARAMETER constants, separated by commas, corresponding to elements in the vlist. An array name in the vlist demands that dlist constants be supplied to fill every element of the array.

Repetition of a constant is provided by using the form:

```
n*constant-value
```

n

a positive integer, is the repetition count.

## Example

```
REAL A, B, C(3), D(2)
DATA A, B, C(1), D /1.0, 2.0, 3.0, 2*4.0/
```

This performs the following initialization:

```
A = 1.0
B = 2.0
C(1) = 3.0
D(1) = 4.0
D(2) = 4.0
```

## DEALLOCATE

The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable array that appears in the statement to be deallocated (freed). Deallocate also deallocates storage for deferred-shape arrays.

F77

### Syntax

```
DEALLOCATE ( allocate-object-list [ , STAT= var ] )
```

Where:

allocate-object-list

is a variable name or a structure component.

al

is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.

var is

the status indicator, an integer variable, integer array element or an integer member of a structure.

### Description

An attempt to deallocate a pointer-based variable or an allocatable COMMON block which was not created by an ALLOCATE statement results in an error condition.

## DECODE

If the specifier `STAT=` is present, successful execution of the statement causes `var` to be defined with the value of zero. If an error occurs during the execution of the statement and the specifier `STAT=` is present, the status variable is defined to have the integer value one. If an error occurs and the specifier `STAT=` is not present, program execution is terminated.

### Examples

```
REAL, ALLOCATABLE :: X(:, :)
ALLOCATE (X(10,2))
X=0
DEALLOCATE (X)
```

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

## DECODE

The `DECODE` statement transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential `READ` statements.

### F77 extension

#### Syntax

```
DECODE (c, f, b [ ,IOSTAT= ios ] [,
ERR= errs]) [ list ]
```

**c**

is an integer expression specifying the number of bytes involved in translation.

**f**

is the format-specifier.

**b**

is a scalar or array reference for the buffer area containing formatted data (characters).

**ios**

is an integer scalar memory reference which is the input/output status specifier: if this is specified `ios` becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

**errs**

an error specifier which takes the form of a statement label of an executable statement in the same program unit. If an error condition occurs execution continues with the statement specified by `errs`.

list

is a list of input items.

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. In the following example, `sum` is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

The preceding code segment is roughly equivalent to this:

```
character*4 ch
ch = '()'
accept ch
```

## See Also

[“READ”](#), [“PRINT”](#)

## DIMENSION

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

## F90

### Syntax

```
DIMENSION [::] array-name (array-spec)
[, array-name (array-spec) ] ...
DIMENSION array-name ([lb:]ub[, [lb:]ub]...)
[, name([lb:]ub[, [lb:]ub]...)]
```

**array-name**

is the symbolic name of an array.

**array-spec** is

a valid array specification, either explicit-shape, assumed-shape, deferred-shape, or assumed size. For more information on array specifications, refer to [Chapter 4, “Fortran Arrays”](#).

**lb:ub**

is a dimension declarator specifying the bounds for a dimension (the lower bound `lb` and the upper bound `ub`). `lb` and `ub` must be integers with `ub` greater than `lb`. The lower bound `lb` is optional; if it is not specified, it is taken to be 1.

### Description

DIMENSION can be used in a subroutine to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays). Note however that these integer variables must be either arguments or declared in COMMON; they

## DO (Iterative)

cannot be local. In this case the function of DIMENSION is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

The default typing of the array in a DIMENSION statement is defined by the initial letter of the array name in the same way as variable names. The letters I,J,K,L,M and N imply that the array is of INTEGER type and an array with a name starting with any of the letters A to H and O to Z will be of type REAL, unless overridden by an IMPLICIT or type declaration statement. Arrays may appear in type declaration and COMMON statements but the array name can appear in only one array declaration.

DIMENSION statements must not be labeled.

## Examples

```
DIMENSION ARRAY1(3:10), ARRAY2(3,-2:2)
```

This specifies ARRAY1 as a vector having eight elements with the lower bound of 3 and the upper bound of 10.

ARRAY2 as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

```
CHARACTER B(0:20)*4
```

This example sets up an array B with 21 character elements each having a length of four characters. Note that the character array has been dimensioned in a type declaration statement and therefore cannot subsequently appear in a DIMENSION statement.

## DO (Iterative)

The DO statement introduces an iterative loop and specifies the loop control index and parameters. There are two forms of DO statement, block and non-block (FORTRAN 77 style). There are two forms of block do statements, DO iterative and DO WHILE. Refer to the description of DO WHILE for more details on this form of DO statement.

## F90

### Syntax

#### DO (block)

```
[do-construct-name : ] DO [label ] [index=e1,e2 [,e3]]  
[execution-part-construct]  
[label] END DO
```

label

labels the end do.

index

is the name of a variable called the DO variable.

e1

is an expression which yields an initial value for i.

e2

is an expression which yields a final value for i.

$e_3$

is an optional expression yielding a value specifying the increment value for  $i$ . The default for  $e_3$  is 1.

## DO (non-block)

```
DO label [,] index = e1, e2 [,
e3]
```

label

labels the last executable statement in the loop (this must not be a transfer of control).

index

is the name of a variable called the DO variable.

$e_1$

is an expression which yields an initial value for  $i$ .

$e_2$

is an expression which yields a final value for  $i$ .

$e_3$

is an optional expression yielding a value specifying the increment value for  $i$ . The default for  $e_3$  is 1.

## Description

The DO loop consists of all the executable statements after the specifying DO statement up to and including the labeled statement, called the terminal statement. The label is optional. If omitted, the terminal statement of the loop is an END DO statement.

Before execution of a DO loop, an iteration count is initialized for the loop. This value is the number of times the DO loop is executed, and is:

```
INT((e2-e1+e3)/e3)
```

If the value obtained is negative or zero the loop is not executed.

The DO loop is executed first with  $i$  taking the value  $e_1$ , then the value  $(e_1+e_3)$ , then the value  $(e_1+e_3+e_3)$ , etc.

It is possible to jump out of a DO loop and jump back in, as long as the do index variable has not been adjusted. In a nested DO loop, it is legal to transfer control from an inner loop to an outer loop. It is illegal, however, to transfer into a nested loop from outside the loop.

## Syntax Extension

Nested DO loops may share the same labeled terminal statement if required. They may not share an END DO statement.

## Examples

```
DO 100 J = -10,10
DO 100 I = -5,5
100 SUM = SUM + ARRAY (I,J)
DO
A(I)=A(I)+1
```

```

IF (A(I).EQ.4) EXIT
END DO
DO I=1,N
A(I)=A(I)+1
END DO

```

## See Also

[“DO WHILE”](#), [“END DO ”](#)

## DO WHILE

The DO WHILE statement introduces a logical do loop and specifies the loop control expression.

The DO WHILE statement executes for as long as the logical expression continues to be true when tested at the beginning of each iteration. If expression is false, control transfers to the statement following the loop.

## F77

## Syntax

```

DO [label[,]] WHILE
expression

```

The end of the loop is specified in the same way as for an iterative loop, either with a labeled statement or an END DO.

**label**

labels the last executable statement in the loop (this must not be a transfer of control).

**expression**

is a logical expression and label.

## Description

The logical expression is evaluated. If it is `.FALSE.`, the loop is not entered. If it is `.TRUE.`, the loop is executed once. Then logical expression is evaluated again, and the cycle is repeated until the expression evaluates `.FALSE.`

## Example

```

DO WHILE (K == 0)
SUM = SUM + X
END DO

```

## See Also

[“DO \(Iterative\)”](#), [“END DO ”](#)

## DOUBLE COMPLEX

The DOUBLE COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type. This overrides the data typing implied by the initial letter of a symbolic name.



## F77 extension

### Syntax

The syntax for `DOUBLE COMPLEX` has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE COMPLEX [, attribute-list ::] entity-list
```

`attribute-list`

is the list of attributes for the double complex variable.

`entity-list`

is the list of defined entities.

### Syntax Extension

```
DOUBLE COMPLEX name [/clist/] [,name] [/clist/]...
```

`name`

is the symbolic name of a variable, array, or an array declarator (see the `DIMENSION` statement for an explanation of array declarators).

`clist`

is a list of constants that initialize the data, as in a `DATA` statement.

### Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the `DIMENSION` statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, `IMPLICIT` statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An `IMPLICIT` statement overrides predefined typing rules.

The default size of a `DOUBLE COMPLEX` variable is 16 bytes. With the `-r8` option, the default size of a `DOUBLE COMPLEX` variable is also 16 bytes.

### Examples

```
DOUBLE COMPLEX CURRENT, NEXT
```

### See Also

[“COMPLEX”](#)

## DOUBLE PRECISION

The `DOUBLE PRECISION` statement establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type. This overrides the data typing implied by the initial letter of a symbolic name.

ELSE

F90

## Syntax

The syntax for DOUBLE PRECISION has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE PRECISION [, attribute-list ::] entity-list
```

attribute-list

is the list of attributes for the double precision variable.

entity-list

is the list of defined entities.

## Syntax Extension

```
DOUBLE PRECISION name [/clist/] [,  
name] [/clist/]....
```

name

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist

is a list of constants that initialize the data, as in a DATA statement.

## Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a DOUBLE PRECISION variable is 8 bytes, with or without the `-r8` option.

## Example

```
DOUBLE PRECISION PLONG
```

## ELSE

The ELSE statement begins an ELSE block of an IF block and encloses a series of statements that are conditionally executed.

F77

## Syntax

```
IF logical expression THEN  
  statements  
ELSE IF logical expression THEN  
  statements  
ELSE
```

```
statements
END IF
```

The ELSE section is optional and may occur only once. Other IF blocks may be nested within the statements section of an ELSE block.

## Example

```
IF (I.LT.15) THEN
  M = 4
ELSE
  M=5
END IF
```

## See Also

[“ELSE IF”](#), [“END IF”](#), [“END IF”](#)

## ELSE IF

The ELSE IF statement begins an ELSE IF block of an IF block series and encloses statements that are conditionally executed.

## F77

## Syntax

```
IF logical expression THEN
  statements
ELSE IF logical expression THEN
  statements
ELSE
  statements
END IF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an ELSE IF block.

## Example

```
IF (I.GT.70) THEN
  M=1
ELSE IF (I.LT.5) THEN
  M=2
ELSE IF (I.LT.16) THEN
  M=3
END IF
```

## See Also

[“ELSE”](#), [“END IF”](#), [“END IF”](#)

## ELSE WHERE

The ELSE WHERE portion of the WHERE ELSE WHERE construct permits conditional masked assignments to the elements of an array or to a scalar, zero-dimensional array.

## ENCODE

### F90

#### Syntax

```
WHERE (logical-array-expr)
  array-assignments
[ELSE WHERE
  array-assignments ]
END WHERE
```

#### Description

The WHERE statement and the WHERE ELSE WHERE construct permit masked assignments to the elements of an array or to a scalar, zero dimensional array. The ELSE WHERE portion of the WHERE ELSE WHERE construct allows you to conditionally mark assignments to elements of an array or to a scalar, zero-dimensional array.

#### Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 )
  GRADE = 'P'
ELSE WHERE
  GRADE = 'F'
END WHERE
```

#### See Also

[“ELSE”](#), [“END IF”](#), [“END WHERE”](#), [“WHERE”](#)

## ENCODE

The ENCODE statement transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE statements.

### F77 extension

#### Syntax

```
ENCODE (c,f,b[,IOSTAT=ios] [,ERR=errs])[list]
```

c

is an integer expression specifying the number of bytes involved in translation.

f

is the format-specifier.

b

is a scalar or array reference for the buffer area receiving formatted data (characters).

ios

is an integer scalar memory reference which is the input/output status specifier: if this is included, ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

**errs**

an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by errs.

**list**

a list of output items.

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4
ch
ch = '()'
accept ch
```

## See Also

[“READ”](#), [“PRINT”](#)

## END

The END statement terminates a segment of a Fortran program. There are several varieties of the END statement, each described in the sections that follow.

## F77

### Syntax

```
END
```

### Description

The END statement terminates a programmed module. The END statement has the same effect as a RETURN statement in a SUBROUTINE or FUNCTION, or the effect of a STOP statement in a PROGRAM program unit. END may be the last statement in a compilation or it may be followed by a new program unit or module.

## END DO

The END DO statement terminates a DO or DO WHILE loop.

## F77

### Syntax

```
END DO
```

## Description

The END DO statement terminates an indexed DO or DO WHILE statement which does not contain a terminal-statement label.

The END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement contains a terminal-statement label.

## See Also

[“DO \(Iterative\)”](#), [“DO WHILE”](#)

## END FILE

The END FILE statement writes an endfile record to the files.

## F77

## Syntax

```
END FILE u
END FILE ([UNIT=]u, [,IOSTAT =ios] [,ERR=errs] )
```

u

is the external unit specifier where u is an integer.

IOSTAT=ios

an integer scalar memory reference which is the input/output specifier: if this is included in list, ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

ERR=errs

an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by errs.

## Description

When an END FILE statement is executed, an endfile record is written to the file as the next record. The file is then positioned after the endfile record. Note that only records written prior to the endfile record can be read later.

A BACKSPACE or REWIND statement must be used to reposition the file after an END FILE statement prior to the execution of any data transfer statement. A file is created if there is an END FILE statement for a file connected but not in existence.

For example:

```
END FILE(20)
END FILE(UNIT=34, IOSTAT=IOERR, ERR=140)
```

## END IF

The END IF statement terminates an IF ELSE or ELSE IF block.

F77

## Syntax

```
END IF
```

## Description

The END IF statement terminates an IF block. Earlier in the same subprogram, there must be a matching block IF statement at the same IF level. For more information, refer to the IF statement.

## Example

```
IF ( I.GT.70 ) THEN
  M=1
ELSE IF ( I.LT.5 ) THEN
  M=2
ELSE IF ( I.LT.16 ) THEN
  M=3
END IF
```

**END MAP**

The END MAP statement terminates a MAP declaration.

## F77 extension

## Syntax

```
END MAP
```

## Description

For more information, refer to the [“MAP”](#) statement.

## Example

```
MAP ! Structure for a customer
  INTEGER*4 acct_cust
  REAL*4 credit_amt
  CHARACTER*8 due_date
END MAP
```

**END SELECT**

The END SELECT statement terminates a SELECT declaration.

F77

## Syntax

```
END SELECT
```

## END STRUCTURE

### Description

For more information, refer to the [“SELECT CASE”](#) statement.

### Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
  TYPE=1
CASE ( 4:6 )
  TYPE=2
CASE DEFAULT
  TYPE=0
END SELECT
```

## END STRUCTURE

The END STRUCTURE statement terminates a STRUCTURE declaration.

### F77 extension

### Syntax

```
END STRUCTURE
```

### Description

For more information, refer to the [“STRUCTURE”](#) statement.

## END UNION

The END UNION statement terminates a UNION declaration.

### F77 extension

### Syntax

```
END UNION
```

### Description

For more information, refer to the [“UNION”](#) statement.

## END WHERE

The END WHERE statement terminates a WHERE ELSE WHERE construct.

### F77 extension

### Syntax

```
END WHERE
```



## Description

For more information, refer to the “[WHERE](#)” statement.

## ENTRY

The ENTRY statement allows a subroutine or function to have more than one entry point.

## F77

## Syntax

```
ENTRY name [(variable, variable...)]
```

### name

is the symbolic name, or entry name, by which the subroutine or function may be referenced.

### variable

is a dummy argument. A dummy argument may be a variable name, array name, dummy procedure or, if the ENTRY is in a subroutine, an alternate return argument indicated by an asterisk. If there are no dummy arguments, name may optionally be followed by (). There may be more than one ENTRY statement within a subroutine or function, but they must not appear within a block IF or DO loop.

## Description

The ENTRY statement provides a method to provide several procedures in a single subprogram, such as additional Function statements in a Function subprogram or additional Subroutine statements in a subroutine subprogram. An ENTRY statement is called in exactly the same manner as a subroutine or function, depending on in which type of construct it appears

There are a number of rules that govern ENTRY statements:

- The name of an ENTRY must not be used as a dummy argument in a FUNCTION, SUBROUTINE or ENTRY statement, nor may it appear in an EXTERNAL statement.
- Within a function a variable name which is the same as the entry name may not appear in any statement that precedes the ENTRY statement, except in a type statement.
- If name is of type character the names of each entry in the function and the function name must be of type character. If the function name or any entry name has a length of (\*) all such names must have a length of (\*); otherwise they must all have a length specification of the same integer value.
- A name which is used as a dummy argument must not appear in an executable statement preceding the ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE or ENTRY statement that precedes the executable statement. Neither must it appear in the expression of a statement function unless the name is also a dummy argument of the statement function, or appears in a FUNCTION or SUBROUTINE statement, or in an ENTRY statement that precedes the statement function statement.
- If a dummy argument appears in an executable statement, execution of that statement is only permitted during the execution of a reference to the function or subroutine if the dummy argument appears in the dummy argument list of the procedure name referenced.

- If name is of type character the names of each entry in the function and the function name must be of type character. If the function name or any entry name has a length of (\*) all such names must have a length of (\*); otherwise they must all have a length specification of the same integer value.
- A name which is used as a dummy argument must not appear in an executable statement preceding the ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE or ENTRY statement that precedes the executable statement. Neither must it appear in the expression of a statement function unless the name is also a dummy argument of the statement function, or appears in a FUNCTION or SUBROUTINE statement, or in an ENTRY statement that precedes the statement function statement.
- If a dummy argument appears in an executable statement, execution of that statement is only permitted during the execution of a reference to the function or subroutine if the dummy argument appears in the dummy argument list of the procedure name referenced.
- When a subroutine or function is called using the entry name, execution begins with the statement immediately following the ENTRY statement. If a function entry has no dummy arguments the function must be referenced by name() but a subroutine entry without dummy arguments may be called with or without the parentheses after the entry name.
- Recursive referencing is not allowed.
- The order, type, number, and names of dummy arguments in an ENTRY statement can be different from those used in the FUNCTION, SUBROUTINE or other ENTRY statements in the same program unit but each reference must use an actual argument list which agrees in order, type, and number with the dummy argument list of the corresponding FUNCTION, SUBROUTINE or ENTRY statement. When a subroutine name or an alternate return specifier is used as an actual argument, there is no need to match the type.
- Entry names within a FUNCTION subprogram need not be of the same data type as the function name, but they all must be consistent within one of the following groups of data types:

```

BYTE, INTEGER*2, INTEGER*4, LOGICAL*1
LOGICAL*2, LOGICAL*4, REAL*4, REAL*8
COMPLEX*8
COMPLEX*16
CHARACTER

```

- If the function is of character data type, all entry names must also have the same length specification as that of the function.

## Example

```

FUNCTION SUM(TALL, SHORT, TINY)
.
SUM=TALL- (SHORT+TINY)
RETURN
ENTRY SUM1(X, LONG, TALL, WIDE, NARROW)
.
.
SUM1= (X*LONG) + (TALL*WIDE) +NARROW
RETURN
ENTRY SUM2(SHORT, SMALL, TALL, WIDE)
.
.
SUM2= (TALL-SMALL) + (WIDE-SHORT)
RETURN
END

```

When the calling program calls the function SUM it can do so in one of three ways depending on which ENTRY point is desired.

For example, for the following call, the ENTRY point is SUM2.

```
Z=SUM2(LITTLE, SMALL, BIG, HUGE)
```

As another example, for the call that follows, the ENTRY point is SUM.

```
Z=SUM(T, X, Y)
```

## EQUIVALENCE

The EQUIVALENCE statement allows two or more named regions of data memory to share the same start address. Arrays that are subject to the EQUIVALENCE statement in HPF are treated as sequential and any attempt at non-replicated data distribution or mapping is ignored for such arrays.

F77

### Syntax

```
EQUIVALENCE (list)[, (list)...]
```

list

is a set of identifiers (variables, arrays or array elements) which are to be associated with the same address in data memory. The items in a list are separated by commas, and there must be at least two items in each list. When an array element is chosen, the subscripts must be integer constants or integer PARAMETER constants.

### Description

You use the EQUIVALENCE statement to make a single region of data memory have different types, such as treating the imaginary part of a complex number as a real value. You can also use this statement to make arrays overlap, so that the same region of storage can be dimensioned in several different ways. Records and record fields cannot be specified in EQUIVALENCE statements.

### Syntax Extension

An array element may be identified with a single subscript in an EQUIVALENCE statement even though the array is defined to be a multidimensional array. Further, EQUIVALENCE of character and non-character data is allowed as long as misalignment of non-character data does not occur.

### Example

```
COMPLEX NUM
REAL QWER(2)
EQUIVALENCE (NUM, QWER(1))
```

In the preceding example, QWER(1) is the real part of NUM and QWER(2) is the imaginary part.

#### Note

EQUIVALENCE statements are illegal if there is any attempt to make a mapping of data memory inconsistent with its linear layout.

## EXIT

The EXIT statement interrupts a DO construct execution and continues with the next statement after the loop.

F90

### Syntax

```
EXIT [do-construct-name]
```

### Example

```
DO
  IF (A(I).EQ.0) CYCLE
  B=100/A(I)
  IF (B.EQ.5) EXIT
END DO
```

### See Also

[“CYCLE”](#), [“DO \(Iterative\)”](#)

## EXTERNAL

The EXTERNAL statement identifies a symbolic name as an external or dummy procedure. This procedure can then be used as an actual argument.

F77

### Syntax

```
EXTERNAL proc [,proc]..
```

proc

is the name of an external procedure, dummy procedure or block data program unit. When an external or dummy procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit.

### Description

If an intrinsic function appears in an EXTERNAL statement an intrinsic function of the same name cannot then be referenced in the program unit. A symbolic name can appear only once in all the EXTERNAL statements of a program unit.

## EXTRINSIC

The EXTRINSIC statement identifies a symbolic name as an external or dummy procedure that is written in some language other than HPF.

HPF

### Syntax

```
EXTRINSIC ( extrinsic-kind-keyword ) procedure name
```

**extrinsic-kind-keyword**

is the name of an extrinsic interface supported. The currently supported value is are HPF\_LOCAL, and F77\_LOCAL.

**procedure name**

is either a subroutine-statement or a function-statement defining a name for an external and extrinsic procedure.

## Description

The EXTRINSIC procedure can then be used as an actual argument once it is defined. The call to an EXTRINSIC procedure should be semantically equivalent to the execution of an HPF procedure in that on return from the procedure, all processors are still available, and all data and templates will have the same distribution and alignment as when the procedure was called.

## See Also

For a complete description of the PGHPF extrinsic facility, along with examples, refer to the PGHPF User's Guide available online at [www.pgroup.com/doc/pghpf\\_ug/hpfug.htm](http://www.pgroup.com/doc/pghpf_ug/hpfug.htm).

## FORALL

The FORALL statement and the FORALL construct provide a parallel mechanism to assign values to the elements of an array.

## F95

### Syntax

```
FORALL (forall-triplet-spec-list [, scalar-mask-expr] )
forall-assignment
```

or

```
FORALL (forall-triplet-spec-list [, scalar-mask-expr] )
forall-body
[forall-body ]...
END FORALL
```

where forall-body is one of:

```
forall-assignment
where-statement
where-construct
forall-statement
forall-construct
```

## Description

The FORALL statement is computed in four stages:

1. Compute the valid set of index values.

Compute the active set of index values, taking into consideration the scalar-mask-expr. If no scalar-mask-expr is present, the valid set is the same as the active set of index values.

## FORMAT

Third, for each index value, the right-hand side of the body of the FORALL is computed. Finally, the right-hand side is assigned to the left-hand side, for each index value.

2. Compute the valid set of index values.

Compute the active set of index values, taking into consideration the scalar-mask-expr. If no scalar-mask-expr is present, the valid set is the same as the active set of index values.

3. For each FORALL statement, in order, compute the right hand side of the assignment or the expression for each active value.

For each index value, the right-hand side of the body of the FORALL is computed.

4. For each FORALL assignment, the right-hand side is assigned to the left-hand side for each active index value.

## Examples

```
FORALL ( I = 1:3 ) A(I) = B(I)
FORALL ( I = 1:L, A(I) == 0.0 ) A(I) = R(I)
FORALL ( I = 1:3 )
  A(I) = D(I)
  B(I) = C(I) * 2
END FORALL
FORALL ( I = 1:5 )
  WHERE ( A(I,:) /= 0.0 )
  A(I,:) = A(I-1,:) + A(I+1,:)
  ELSEWHERE
  B(I,:) = A(6-I,:)
  END WHERE
END FORALL
```

## FORMAT

The FORMAT statement specifies format requirements for input or output.

## F77

## Syntax

```
label FORMAT (list-items)
```

### list-items

can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant that defines the number of repeats.
- Nonrepeatable editor commands.
- A format specification list optionally preceded by an integer constant that defines the number of repeats.

Each action of format control depends on the next edit code and the next item in the input/output list where one is used. If an input/output list contains at least one item there must be at least one repeatable edit code

in the format specification. An empty format specification ( ) can only be used if no list items are specified; in such a case one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. Where a repeat specification is used the appropriate item is repeated the required number of times.

## Description

For details on using the FORMAT statement, refer to [Chapter 5, “Input and Output”](#).

## Examples

```
WRITE (6,90) NPAGE
90 FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

This code produces:

```
PAGE NUMBER 10 SALES REPORT, Cont.
```

The following example shows use of the tabulation specifier T:

```
PRINT 25
25 FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

This code produces:

```
                COLUMN 1                COLUMN 2
```

The following example shows use of formatting with an array:

```
DIMENSION A(6)
DO 10 I = 1,6
10 A(I) = 25.
TYPE 100,A
100 FORMAT(' ',F8.2,2PF8.2,F8.2) ! ' '
C ! gives single spacing
```

produces:

```
25.00
2500.00 2500.00
2500.00 2500.00 2500.00
```

## Note

The effect of the scale factor continues until another scale factor is used.

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. In the following example, sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4
ch
ch = '()'
accept ch
```

## See Also

[“READ”](#), [“PRINT”](#)

## FUNCTION

The **FUNCTION** statement introduces a program unit; the statements that follow all apply to the function itself and are laid out in the same order as those in a **PROGRAM** program unit.

## F77

### Syntax

```
[function-prefix] FUNCTION name [*n] ([argument [,argument]...])
...
END [ FUNCTION [function-name]]
```

#### function-prefix

is one of:

[type-spec] RECURSIVE

[RECURSIVE ] type-spec

where type-spec is a valid type specification. Type will explicitly apply a type to the function. If the function is not explicitly typed then the function type is taken from the initial letter and is dictated by the usual default.

#### name

is the name of the function and must be unique among all the program unit names in the program. name must not clash with any local, COMMON or PARAMETER names.

#### \*n

is the optional length of the data type.

#### argument

is a symbolic name, starting with a letter and containing only letters and digits. An argument can be of type REAL, INTEGER, DOUBLE PRECISION, CHARACTER, LOGICAL, COMPLEX, or BYTE, and so on.

### Description

**FUNCTION** statements and names apply only to the function, except for subroutine or function references and the names of COMMON blocks. These rules govern the **FUNCTION** statement:

- The function must be terminated by an **END** statement.
- A function produces a result; this allows a function reference to appear in an expression, where the result is assumed to replace the actual reference.
- The symbolic name of the function must appear as a variable in the function, unless the **RESULT** keyword is used. The value of this variable, on exit from the function, is the result of the function. The function result is undefined if the variable has not been defined.



- The type of a FUNCTION refers to the type of its result.
- Recursion is allowed if you use the `-Mrecursive` option on the command-line or if the `RECURSIVE` prefix is included in the function definition.

## Examples

```

FUNCTION FRED(A,B,C)
REAL X
.
END
FUNCTION EMPTY() ! Note parentheses
END
PROGRAM FUNCALL
.
SIDE=TOTAL(A,B,C)
.
END
FUNCTION TOTAL(X,Y,Z)
.
END
FUNCTION AORB(A,B)
IF(A-B)1,2,3
1 AORB = A
RETURN
2 AORB = B
RETURN
3 AORB = A + B
RETURN
END

```

## See Also

[“PURE”](#), [“RECURSIVE”](#), [“RETURN”](#)

## GOTO (Assigned)

(Obsolescent) The assigned GOTO statement transfers control so that the statement identified by the statement label is executed next. Internal procedures can be used in place of the assign statement used with an assigned GO TO.

## F77

## Syntax

```
GOTO integer-variable-name[[,] (list)]
```

### integer-variable-name

must be defined with the value of a statement label of an executable statement within the same program unit. This type of definition can only be done by the ASSIGN statement.

### list

consists of one or more statement labels attached to executable statements in the same program unit. If a list of statement labels is present, the statement label assigned to the integer variable must be in that list.

## Examples

```

ASSIGN 50 TO K
GO TO K(50,90)
90 G=D**5
.
.
50 F=R/T

```

## GOTO (Computed)

The computed GOTO statement allows transfer of control to one of a list of labels according to the value of an expression.

### F77

#### Syntax

```
GOTO (list) [,] expression
```

##### list

is a list of labels separated by commas.

##### expression

selects the label from the list to which to transfer control. Thus a value of 1 implies the first label in the list, a value of 2 implies the second label and so on. An expression value outside the range will result in transfer of control to the statement following the computed GOTO statement.

#### Example

```

READ *, A, B
GO TO (50,60,70)A
WRITE (*, 10) A, B
10 FORMAT (' ', I3, F10.4, 5X, 'A must be 1, 2 or 3')
STOP
50 X=A**B ! Come here if A has the value 1
GO TO 100
60 X=(A*56)*(B/3) ! Come here if A is 2
GO TO 100
70 X=A*B ! Come here if A has the value 3
100 WRITE (*, 20) A, B, X
20 FORMAT (' ', I3, F10.4, 5X, F10.4)

```

## GOTO (Unconditional)

The GOTO statement unconditionally transfers control to the statement with the label `label`. The statement `label` must be declared within the code of the program unit containing the GOTO statement and must be unique within that program unit.

### F77

#### Syntax

```
GOTO label
```

label  
is a statement label

## Example

```
TOTAL=0.0
30 READ *, X
IF (X.GE.0) THEN
TOTAL=TOTAL+X
GOTO 30
END IF
```

## IF (Arithmetic)

(Obsolescent) The arithmetic IF statement transfers control to one of three labeled statements. The statement chosen depends upon the value of an arithmetic expression.

F77

### Syntax

```
IF (arithmetic-expression) label-1, label-2, label-3
```

Control transfers to label-1, label-2 or label-3 if the result of the evaluation of the arithmetic-expression is less than zero, equal to zero or greater than zero respectively.

## Example

```
IF X 10, 20, 30
```

If X is less than zero then control is transferred to label 10. If X equals zero then control is transferred to label 20. If X is greater than zero then control is transferred to label 30.

## IF (Block)

The block IF statement consists of a series of statements that are conditionally executed.

F77

### Syntax

```
IF logical expression THEN
statements
ELSE IF logical expression THEN
statements
ELSE
statements
END IF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an IF block or ELSE IF block.

The ELSE section is optional and may occur only once. Other IF blocks may be nested within the statements section of an ELSE block.

## IF (Logical)

### Example

```
IF ( I.GT.70 ) THEN
  M=1
ELSE IF ( I.LT.5 ) THEN
  M=2
ELSE IF ( I.LT.16 ) THEN
  M=3
END IF
IF ( I.LT.15 ) THEN
  M = 4
ELSE
  M=5
END IF
```

## IF (Logical)

The logical IF statement executes or does not execute a statement based on the value of a logical expression.

### F77

#### Syntax

```
IF (logical-expression) statement
```

#### logical-expression

is evaluated and if it is true the statement is executed. If it is false, the statement is not executed and control is passed to the next executable statement.

#### statement

can be an assignment statement, a CALL statement or a GOTO statement.

#### Examples

```
IF(N .LE. 2) GOTO 27
IF(HIGH .GT. 1000.0 .OR. HIGH .LT. 0.0) HIGH=1000.0
```

## IMPLICIT

The IMPLICIT statement redefines the implied data type of symbolic names from their initial letter. Without the use of the IMPLICIT statement all names that begin with the letters I,J,K,L,M or N are assumed to be of type integer and all names beginning with any other letters are assumed to be real.

### F77

#### Syntax

```
IMPLICIT spec (a[,a]...) [,spec(a[,a]...)]
IMPLICIT NONE
```

#### spec

is a data type specifier.

#### a

is an alphabetic specification expressed either as a or a1-a2, specifying an alphabetically ordered range of letters.

## Description

**IMPLICIT** statements must not be labeled.

Symbol names may begin with a dollar sign (\$) or underscore (\_) character, both of which are of type **REAL** by default. In an **IMPLICIT** statement, these characters may be used in the same manner as other characters, but they cannot be used in a range specification.

The **IMPLICIT NONE** statement specifies that all symbolic names must be explicitly declared, otherwise an error is reported. If **IMPLICIT NONE** is used, no other **IMPLICIT** can be present.

## Examples

```
IMPLICIT REAL (L,N)
IMPLICIT INTEGER (S,W-Z)
IMPLICIT INTEGER (A-D,$,_)
```

## IMPORT

The **IMPORT** statement, used only in an interface body, gives access to the named entities of the containing scope.

## F2003

## Syntax

```
IMPORT [ [::] import-name-list ]
```

::

is a data type specifier.

**import-name-list**

is a list of **import-names**, each of which is an entity that is accessible in the containing scope.

If no **import-name-list** exists, the **IMPORT** statement imports all entities from the containing scope that are not declared as local entities.

## Description

**IMPORT** statements allow the user to access entities accessible in the containing scope of the interface body. Any import entity defined in the containing scope unit must be explicitly declared prior to the interface body. If no import entities are listed, the **IMPORT** statement imports all entities from the containing scope unit except those declared to be local entities.

The **IMPORT** statement must appear after any **USE** statement and prior to any other statements in the body.

## Example

```
interface
  function myfunc(h)
    import    :: x, y
    type(x)  :: myfunc
```

## INCLUDE

```
real(y) :: h
end function myfunc
end interface
```

## INCLUDE

The INCLUDE statement directs the compiler to start reading from another file.

### Note

---

The INCLUDE statement is used for FORTRAN 77. There is no support for VAX/VMS text libraries or the `module_name` pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

## F77 extension

### Syntax

```
INCLUDE 'filename [/[NO]LIST]'
INCLUDE "filename [/[NO]LIST]"
```

The following rules apply to the INCLUDE statement:

- The INCLUDE statement may be nested to a depth of 20 and can appear anywhere within a program unit as long as Fortran's statement-ordering restrictions are not violated.
- You can use the qualifiers `/LIST` and `/NOLIST` to control whether the include file is expanded in the listing file (if generated).

### Note

---

There is no support for VAX/VMS text libraries or the `module_name` pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

- Either single or double quotes may be used.
- If the final component of the file pathname is `/LIST` or `/NOLIST`, the compiler assumes it is a qualifier, unless an additional qualifier is supplied.
- The filename and the `/LIST` or `/NOLIST` qualifier may be separated by blanks.

The compiler searches for the include file in the following directories:

- Each `-I` directory specified on the command-line.
- The directory containing the file that contains the INCLUDE statement (the current working directory.)
- The standard include area.

### Example

```
INCLUDE '/mypath/list /list'
```

This line includes a file named `/mypath/list` and expands it in the listing file, if a listing file is used.

## INQUIRE

An INQUIRE statement has two forms and is used to inquire about the current properties of a particular file or the current connections of a particular unit. INQUIRE may be executed before, during or after a file is connected to a unit.

F77

### Syntax

```
INQUIRE (FILE=filename, list)
INQUIRE ([UNIT=]unit, list)
```

In addition list may contain one of each of the following specifiers in any order, following the unit number if the optional UNIT specifier keyword is not supplied.

ACCESS=acc

acc returns a character variable which is defined according to the access method for the file as either DIRECT or SEQUENTIAL.

ACTION=acc

acc is a character variable which is defined according to the access types for the connection: READ, WRITE, or READWRITE.

BLANK= blk

blk is a character variable which is defined as NULL, ZERO, or UNDEFINED. NULL means all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of zero. ZERO means all blanks other than leading blanks to be treated as zeros. This specifier must only be used when a file is connected for formatted input/output. UNDEFINED is returned if there is no connection or if the connection is not for formatted I/O.

DELIM= del\_char del\_char

is a character values which is defined as the value APOSTROPHE, QUOTE, NONE or UNDEFINED. These values specify the character delimiter for list-directed or namelist formatted data transfer statements according to the value used on the corresponding OPEN statement.

DIRECT= dir\_char

dir\_char is a character reference which is defined as the value YES, NO, or UNKNOWN. YES means DIRECT is one of the allowed access methods for the file. NO means DIRECT is not an allowed access method for the file. UNKNOWN means it is not known if DIRECT is included in the allowed access methods.

ERR= errs

errs is an error specifier which returns the value of a statement label of an executable statement within the same program. If an error condition occurs execution continues with the statement specified by errs.

EXIST= value

value is a logical variable or logical array element which becomes .TRUE. if there is a file/unit with the name or .FALSE. otherwise.

FILE= fin

fin is a character expression whose value is the file name expression, the name of the file connected to the specified unit.

FORM= fm

fm is a character expression specifying whether the file is being connected for FORMATTED or UNFORMATTED input/output. The default is UNFORMATTED.

FORMATTED= fmt

fmt is a character memory reference which takes the value YES if FORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if FORMATTED is included.

IOSTAT= ios

ios input/output status specifier where ios is an integer reference: if this is included in list, ios is defined as 0 if no error condition occurred and a positive integer when there is an error condition.

NAME= fn

fn is a character scalar memory reference which is assigned the name of the file when the file has a name, otherwise it is undefined

NAMED= nmd

nmd is a logical scalar memory reference which becomes .TRUE. if the file has a name, otherwise it becomes .FALSE.

NEXTREC= nr

nr is an integer scalar memory reference which is assigned the value n+1, where n is the number of the record read or written. It takes the value 1 if no records have been read or written. If the file is not connected or its position is indeterminate nr is undefined.

NUMBER= num

num is an integer scalar memory reference or integer array element assigned the value of the external unit number of the currently connected unit. It becomes undefined if no unit is connected.

OPENED= od

od is a logical scalar memory reference which becomes .TRUE. if the file/unit specified is connected (open) and .FALSE. if the file is not connected (.FALSE.).

PAD= pad\_char

pad\_char is a character expression specifying whether to use blank padding. Values for pad\_char are YES or NO: yes specifies blank padding is used, no requires that input records contain all requested data.

PENDING= pend\_char

pend\_char is a character expression specifying whether previously pending asynchronous data transfers are complete.

A data transfer operation is previously pending if it is pending at the beginning of execution of the INQUIRE statement.

- If an ID= specifier appears and the specified data transfer operation is complete, then the variable specified in the PENDING= specifier is assigned the value FALSE and the INQUIRE statement performs the 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, then the variable specified in the PENDING= specifier is assigned the value FALSE and the INQUIRE statement performs wait operations for all previously pending data transfers for the specified unit.



- In all other cases, the variable specified in the `PENDING=` specifier is assigned the value `TRUE` and no wait operations are performed. The previously pending data transfers remain pending after the execution of the `INQUIRE` statement.

### Note

---

The processor has considerable flexibility in defining when it considers a transfer to be complete.

A processor can use any of the following approaches to determine when a transfer is complete:

- The `INQUIRE` statement could consider an asynchronous data transfer to be incomplete until after the corresponding wait operation. In this case `PENDING=` would always return true unless there were no previously pending data transfers for the unit.
- The `INQUIRE` statement could wait for all specified data transfers to complete and then always return false for `PENDING=`.
- The `INQUIRE` statement could actually test the state of the specified data transfer operations.

`POS= pos_char`

`pos_char` is a scalar-int variable that is assigned the number of the file storage unit immediately following the current position of a file connected for stream access.

- If the file is positioned at its terminal position, the variable 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 can not be determined because of previous error conditions, then the variable is undefined.

`POSITION= pos_char`

`pos_char` is a character expression specifying the file position. Values are `ASIS`, `REWIND` or `APPEND`. For a connected file, on `OPEN` `ASIS` leaves the position in the current position, `REWIND` rewinds the file and `APPEND` places the current position at the end of the file, immediately before the end-of-file record.

`READ= rl`

`rl` is a character reference which takes the value `YES` if `UNFORMATTED` is one of the allowed access methods for file, `NO` if not, or `UNKNOWN` if it is not known if `UNFORMATTED` is included.

`READWRITE= rl`

`rl` is a character scalar memory reference which takes the value `YES` if `UNFORMATTED` is one of the allowed access methods for the file, `NO` if not and `UNKNOWN` if it is not known if `UNFORMATTED` is included.

`RECL= rcl`

`rcl` is an integer expression defining the record length in a file connected for direct access. When sequential input/output is specified this is the maximum record length. This specifier must only be given when a file is connected for direct access.

`SEQUENTIAL= seq`

`seq` is a character scalar memory reference which takes the value `YES` if `UNFORMATTED` is one of the allowed access methods for the file, `NO` if not and `UNKNOWN` if it is not known if `UNFORMATTED` is included.

UNFORMATTED= unf

unf is a character scalar memory reference which takes the value YES if UNFORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if UNFORMATTED is included.

WRITE= rl

rl is a character scalar memory reference which takes the value YES, NO, or UNKNOWN. These values indicate that WRITE is allowed, not allowed, or indeterminate for the specified file, respectively.

## Description

When an INQUIRE by file statement is executed the following specifiers will only be assigned values if the file name is acceptable: nmd, fn, seq, dir, fmt and unf. num is defined, and acc, fm, rcl, nr and blnk may become defined only if od is defined as .TRUE..

When an INQUIRE by unit statement is executed the specifiers num, nmd, fn, acc, seq, dir, fm, fmt, unf, rcl, nr and blnk are assigned values provided that the unit exists and a file is connected to that unit. Should an error condition occur during the execution of an INQUIRE statement all the specifiers except ios become undefined.

## INTEGER

The INTEGER statement establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type. This overrides the data typing implied by the initial letter of a symbolic name.

### F77

### Syntax

The syntax for INTEGER has two forms, a standard FORTRAN 77 or 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
INTEGER [( [ KIND = kind-value) ]
        [, attribute-list ::] entity-list
```

INTEGER permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list

is the list of attributes for the character variable.

entity-list

is the list of defined entities.

### Syntax Extension

```
INTEGER [*n] [,] name [*n] [dimensions] [/clist/]...
```

n

is an optional size specification.

name

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist

is a list of constants that initialize the data, as in a DATA statement.

## Description

Integer type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. INTEGER statements must not be labeled. The default size of an INTEGER variable is 4 bytes. With the `-Mnoi4` option, the default size of an INTEGER variable is 2 bytes.

### Note

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

## Example

```
INTEGER TIME, SECOND, STORE (5,5)
```

## INTENT

The INTENT specification statement (attribute) specifies intended use of a dummy argument. This statement (attribute) may not be used in a main program's specification statement.

## F90

## Syntax

```
INTENT (intent-spec) [ :: ] dummy-arg-list
```

intent-spec

is one of:

- IN
- OUT
- INOUT

dummy-arg-list

is the list of dummy arguments with the specified intent.

## Description

With intent specified as IN, the subprogram argument must not be redefined by the subprogram.

With intent specified as OUT, the subprogram should use the argument to pass information to the calling program.

With intent specified as INOUT, the subprogram may use the value passed through the argument, but should also redefine the argument to pass information to the calling program.

## INTERFACE

### See Also

[“OPTIONAL”](#)

### Example

```
SUBROUTINE IN_OUT(R1,I1)
  REAL::R1
  INTENT (IN)::R1
  INTEGER, INTENT(OUT)::I1
  I1=R1
END SUBROUTINE IN_OUT
```

## INTERFACE

The **INTERFACE** statement block makes an implicit procedure an explicit procedure where the dummy parameters and procedure type are known to the calling module. This statement is also used to overload a procedure name.

### F90

### Syntax

```
INTERFACE [generic-spec]
  [interface-body]...
  [MODULE PROCEDURE procedure-name-list]...
END INTERFACE
```

where **generic-spec** is one of the following:

```
generic-name
OPERATOR (defined operator)
ASSIGNMENT (=)
```

and the interface body specifies the interface for a function or a subroutine:

```
function-statement
  [specification-part]
END FUNCTION [function name]
subroutine-statement
  [specification-part]
END SUBROUTINE [subroutine name]
```

### See Also

[“END ”](#), [“INTERFACE”](#)

### Example

```
INTERFACE
  SUBROUTINE IN_OUT(R1,I1)
  REAL, INTENT (IN)::R1
  INTEGER, INTENT(OUT)::I1
  END SUBROUTINE IN_OUT
```

```
END INTERFACE
```

## INTRINSIC

An **INTRINSIC** statement identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.

F77

### Syntax

```
INTRINSIC func [,func]
```

**func**

is the name of an intrinsic function such as SIN, COS, etc.

### Description

Do not use any of the following functions in **INTRINSIC** statements:

- type conversions:

```
INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR
```

- lexical relationships:

```
LGE, LGT, LLE, LLT
```

- values:

```
MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1
```

When a specific name of an intrinsic function is used as an actual argument in a program unit it must appear in an **INTRINSIC** statement in that program unit. If the name used in an **INTRINSIC** statement is also the name of a generic intrinsic function, it retains its generic properties. A symbolic name can appear only once in all the **INTRINSIC** statements of a program unit and cannot be used in both an **EXTERNAL** and **INTRINSIC** statement in a program unit.

The following example illustrates the use of **INTRINSIC** and **EXTERNAL**:

```
EXTERNAL MYOWN
INTRINSIC SIN, COS
...
CALL TRIG (ANGLE, SIN, SINE)
...
CALL TRIG (ANGLE, MYOWN, COTANGENT)
...
CALL TRIG (ANGLE, COS, SINE)
SUBROUTINE TRIG (X, F, Y)
Y=F(X)
RETURN
END
FUNCTION MYOWN
MYOWN=COS(X)/SIN(X)
RETURN
END
```

In this example, when TRIG is called with a second argument of SIN or COS the function reference F(X) references the intrinsic functions SIN and COS; however when TRIG is called with MYOWN as the second argument F(X) references the user function MYOWN.

## LOGICAL

The LOGICAL statement establishes the data type of a variable by explicitly attaching the name of a variable to a logical data type. This overrides the data typing implied by the initial letter of a symbolic name.

F77

### Syntax

The syntax for LOGICAL has two forms, a standard FORTRAN 77 and 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
LOGICAL [ ( [ KIND = kind-value) ]
        [, attribute-list ::] entity-list
```

LOGICAL permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list

is the list of attributes for the character variable.

entity-list

is the list of defined entities.

### Syntax Extension

```
LOGICAL [*n] [,] name [*n] [dimensions] [/clist/]
        [, name] [*n][dimensions] [/clist/]...
```

n

is an optional size specification.

name

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist

is a list of constants that initialize the data, as in a DATA statement.

### Description

Logical type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a LOGICAL variable is 4 bytes. With the `-MnOi4` option, the default size of a LOGICAL variable is 2 bytes.

## Example

```
LOGICAL TIME, SECOND, STORE(5,5)
```

## MAP

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. For more information on field alignment, refer to [“Structures,” on page 28](#).

## F77 extension

### Syntax

```
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 typed data field within a union.

## Description

Data can be initialized in field declaration statements in union declarations. However, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area 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 field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

## Example

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
  INTEGER typetag ! Tag to determine defined map
  UNION
  MAP ! Structure for an employee
  CHARACTER*12 ssn ! Social Security Number
  REAL*4 salary
  CHARACTER*8 empdate ! Employment date
  END MAP
```

```

MAP ! Structure for a customer
INTEGER*4 acct_cust
REAL*4 credit_amt
CHARACTER*8 due_date
END MAP
MAP ! Structure for a supplier
INTEGER*4 acct_supp
REAL*4 debit_amt
BYTE num_items
BYTE items(12) ! Items supplied
END MAP
END UNION
END STRUCTURE
RECORD /account/ recarr(1000)

```

## MODULE

(PGF95 and PGHPF only) The **MODULE** statement specifies the entry point for a Fortran 90/95 module program unit. A module defines a host environment of scope of the module, and may contain subprograms that are in the same scoping unit.

### F90

#### Syntax

```

MODULE name
  [specification-part]
  [ CONTAINS [module-subprogram-part]]
END [MODULE [ module-name ]]

```

##### name

the name of the program module. This name must not clash with any other names used in the program.

##### specification-part

contains specification statements. See the Fortran 95 Handbook for a complete description of the valid statements.

##### module-subprogram-part

contains function and subroutine definitions for the module, preceded by a single **CONTAINS** keyword.

Modules can be independently compiled and used within programs using the **USE** statement. Use of Fortran 90/95 modules causes the compiler to create a `module_name.mod` file in the current directory (a `.mod` file). This file contains all the information the compiler needs concerning interface specifications and the data types for the routines defined in the module. When a program, routine, or another module encounters the **USE** statement, the `.mod` file is read and "included" in the program, using the scope rules defined in Fortran 90/95 for **USE** association. If you are using separate modules, this creates another step in the program development process. When a module is compiled, both a `.mod` and an object file are created. The `.mod` file is used when a **USE** statement is encountered, and the object file is used when the program is linked.

For example, if `module1.f` contains a module with several procedures, and `test1.f` contains a **USE** statement that uses `module1`, the compilation would involve the steps.

```

% pgf95 -c module1.f
% pgf95 -o test1 test1.f module1.o

```



The search for a .mod file includes the following directories:

- The directory containing the file that contains the USE statement (the current working directory.)
- The directory containing the file that contains the USE statement (the current working directory.)
- Each `-I` directory specified on the command-line.
- The directory specified on the `-moddir` command-line switch.
- The standard include area.

You can use the `-I` command-line option to add directories to the search path for .mod files.

## Example

```
MODULE MYOWN
  REAL MEAN, TOTAL
  INTEGER, ALLOCATABLE, DIMENSION(:):: A
  CONTAINS
  RECURSIVE INTEGER FUNCTION X(Y)
  .
  .
  .
  END FUNCTION X
END MODULE MYOWN
```

## NAMELIST

The NAMELIST statement allows for the definition of namelist groups for namelist-directed I/O.

### F90

### Syntax

```
NAMELIST /group-name/ namelist [[,] /group-name/ namelist ]...
```

**group-name**

is the name of the namelist group.

**namelist**

is the list of variables in the namelist group.

### Example

In the following example the namelist group PERS consists of a name, an account, and a value.

```
CHARACTER*12 NAME
INTEGER*$ ACCOUNT
REAL*4 VALUE
NAMELIST /PERS/ NAME, ACCOUNT, VALUE
```

## NULLIFY

The NULLIFY statement disassociates a pointer from its target.

OPEN

F90

Syntax

```
NULLIFY (pointer-object-list)
```

Example

```
NULLIFY (PTR1)
```

See Also

[“ALLOCATE”](#), [“DEALLOCATE”](#)

## OPEN

The OPEN statement connects an existing file to a unit, creates and connects a file to a unit, creates a file that is preconnected or changes certain specifiers of a connection between a file and a unit.

F77

Syntax

```
OPEN ( list )
```

list

must contain exactly one unit specifier of the form:

```
[UNIT=] u
```

where the UNIT= is optional and the external unit specifier u is an integer.

In addition, list may contain one of each of the following specifiers in any order, following the unit number if the optional UNIT specifier keyword is not supplied.

ACCESS= acc

acc is a character expression specifying the access method for file connection as SEQUENTIAL, DIRECT or STREAM; the default is SEQUENTIAL.

ACTION= acc

acc is a character expression specifying the permitted access types for connection. One of READ, WRITE, UNKNOWN or READWRITE is allowed. The default is UNKNOWN .

ASYNCHRONOUS=async

async is a character expression specifying whether to allow asynchronous data transfer on this file connection. One of 'YES' or 'NO' is allowed.

BLANK=blnk

blnk is a character expression which takes the value 'NULL' or 'ZERO'. 'NULL' causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value

of zero. 'ZERO' causes all blanks other than leading blanks to be treated as zeros. The default is 'NULL.'  
This specifier must only be used when a file is connected for formatted input/output.

**DELIM= del\_char**

del\_char is a character expression which takes the value 'APOSTROPHE', 'QUOTE' or 'NONE'. These values specify the character delimiter for list-directed or namelist formatted data transfer statements.

**ERR=errs**

errs is an error specifier; it takes the form of a statement label of an executable statement within the program. If an error condition occurs execution continues with the statement specified by errs.

**FILE= fin**

fin is a character expression whose value is the file name expression, the name of a file to be connected to the specified unit.

**FORM=fm**

fm is a character expression specifying whether the file is being connected for 'FORMATTED' or 'UNFORMATTED' input/output.

**IOSTAT= ios**

ios is an integer scalar; if this is included ios becomes defined with 0 (zero) if no error condition exists or a positive integer when there is an error condition. A value of -1 indicates an end-of-file condition with no error. A value of -2 indicates an end-of-record condition with no error when using non-advancing I/O.

**PAD= pad\_char**

pad\_char is a character expression specifying whether to use blank padding. Acceptable values are YES or NO; yes specifies that blank padding is used and no requires that input records contain all requested data.

**POSITION= pos\_char**

pos\_char is a character expression specifying the file position. Values are ASIS, REWIND or APPEND. For a connected file, on OPEN ASIS leaves the position in the current position, REWIND rewinds the file and APPEND places the current position at the end of the file, immediately before the end-of-file record.

**RECL= rl**

rl is an integer expression defining the record length in a file connected for direct access. When sequential input/output is specified this is the maximum record length.

**STATUS= sta**

sta is a character expression whose value can be: NEW, OLD, SCRATCH, UNKNOWN or REPLACE. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default status is UNKNOWN which specifies that the file's existence is unknown. If the file exists, the existing file is used; otherwise the file is created. With status OLD, the file must exist or an error is reported. With status NEW, the file is created; if the file exists, an error is reported. Status SCRATCH specifies that the file is removed when closed.

## Description

The record length, RECL=, must be specified if a file is connected for direct access, and optionally, one of each of the other specifiers may be used. RECL is ignored if the access method is sequential.

The unit specified must exist and once connected by an OPEN statement can be referenced in any program unit of the executable program.

## Note

---

If a file is connected to a unit it cannot be connected to a different unit by the OPEN statement.

If a unit is connected to an existing file, execution of another OPEN statement for that file is allowed. Where FILE= is not specified the file to be connected is the same as the file currently connected. If the file specified for connection to the unit does not exist but is the same as a preconnected file, the properties specified by the OPEN statement become part of the connection. However, if the file specified is not the same as the preconnected file this has the same effect as the execution of a CLOSE statement without a STATUS= specifier immediately before the execution of the OPEN statement. When the file to be connected is the same as the file already connected only the BLANK= specifier may be different from the one currently defined.

The sequential and direct access methods both access files that contain fixed-length records. The stream access method, a Fortran 2003 language extension, allows access to files that do not contain fixed-length records. Stream I/O is enabled by specifying access='STREAM'. Stream I/O may be formatted or unformatted.

Asynchronous i/o, the ability to return control before the statement has completed, is supported in certain situations. If ASYNCHRONOUS='yes' is specified on the OPEN statement and a READ or WRITE statement for a particular file connection, a processor may perform an asynchronous data transfer asynchronously, but is not required to do so. In practice, the underlying operating system controls much of what can be performed. A file must be seekable to support asynchronous I/O; i.e. you cannot perform asynchronous I/O on a non-seekable file such as a fifo. Asynchronous I/O is only supported for the stream access method.

## Examples

In the following example a new file, BOOK, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters. Numeric values will have blanks ignored and E1 will be assigned some positive value if an error condition exists when the OPEN statement is executed; execution will then continue with the statement labeled 20. If no error condition pertains, E1 is assigned the value zero (0) and execution continues with the next statement.

```
OPEN(12, IOSTAT=E1, ERR=20, FILE='BOOK',
+ BLANK='NULL', ACCESS='DIRECT', RECL=98,
+ FORM='FORMATTED', STATUS='NEW')
```

The next example shows how to use asynchronous I/O.

```
program test
character*13 b
b = "hello, world\n"
open(unit=10,file='u.dat',access='stream',form='unformatted',
&
asynchronous='yes')
write (unit=10,asynchronous='yes') b
! Do something useful
wait(10)
close(10)
end
```

## Environment Variables

For an OPEN statement which does not contain the FILE= specifier, an environment variable may be used to specify the file to be connected to the unit. If the environment variable FORddd exists, where ddd is a 3 digit string whose value is the unit, the environment variable's value is the name of the file to be opened.

## PGI Fortran Extensions

PGI has extended the OPEN statement as follows:

**CONVERT=order**

order is a character expression specifying the byte order of the file. One of 'BIG\_ENDIAN', 'LITTLE\_ENDIAN', or 'NATIVE' is allowed.

The CONVERT specifier allows byte-swapping I/O to be performed on specific logical units. The value 'BIG\_ENDIAN' is used to convert big-endian format data files produced by most RISC workstations and high-end servers to the little-endian format used on Intel Architecture systems on-the-fly during file reads/writes. This value assumes that the record layouts of unformatted sequential access and direct access files are the same on the systems. For the values 'LITTLE\_ENDIAN' and 'NATIVE', byte-swapping is not performed during file reads/writes since the little-endian format is used by the x86 architecture.

## VAX/VMS Fortran Extensions

VAX/VMS introduced a number of extensions to the OPEN statement. Many of these relate only to the VMS file system and are not supported (e.g., KEYED access for indexed files). The following keywords for the OPEN statement have been added or augmented as shown in the following list. Refer to Programming in VAX FORTRAN for additional details on these keywords that are available for the OPEN statement.

**ACCESS=acc**

The value of 'APPEND' will be recognized and implies sequential access and positioning after the last record of the file. Opening a file with append access means that each appended record is written at the end of the file.

**ASSOCIATEVARIABLE=var**

This keyword specifies an INTEGER\*4 integer scalar memory reference which is updated to the next sequential record number after each direct access I/O operation. Applies only to direct access mode.

**DISPOSE=disp and DISP=disp**

These keywords specify the disposition for the file after it is closed. 'KEEP' or 'SAVE' is the default on anything other than STATUS='SCRATCH' files. 'DELETE' indicates that the file is to be removed after it is closed. The PRINT and SUBMIT values are not supported.

**NAME=file**

This keyword is a synonym for FILE.

**READONLY**

This keyword specifies that an existing file can be read but prohibits writing to that file. The default is read/write.

**RECL=len**

The record length given is interpreted as the number of words in a record if the runtime environment parameter FTNOPT is set to "vaxio". This simplifies the porting of VAX/VMS programs. The default is that len is given in number of bytes in a record.

**TYPE=sta**

This keyword is a synonym for STATUS.

## OPTIONAL

The OPTIONAL specification statement (attribute) specifies dummy arguments that may be omitted or that are optional.

F90

### Syntax

```
OPTIONAL [::] dummy-arg-list
```

### Examples

```
OPTIONAL :: VAR4, VAR5
OPTIONAL VAR6, VAR7
INTEGER, OPTIONAL :: VAR8, VAR9
```

### See Also

[“INTENT”](#)

## OPTIONS

The OPTIONS statement confirms or overrides certain compiler command-line options.

F77 extension

### Syntax

```
OPTIONS /option [/option ...]
```

The following table shows what options are available for the OPTIONS statement:

Table 3.2. OPTIONS Statement

Option	Action Taken
CHECK=ALL	Enable array bounds checking
CHECK=[NO]OVERFLOW	None (recognized but ignored)
CHECK=[NO]BOUNDS	(Disable) Enable array bounds checking
CHECK=[NO]UNDERFLOW	None
CHECK=NONE	Disable array bounds checking
NOCHECK	Disable array bounds checking
[NO]EXTEND_SOURCE	(Disable) Enable the <code>-Mextend</code> option
[NO]G_FLOATING	None
[NO]F77	(Disable) Enable the <code>-Mstandard</code> option
[NO]I4	(Disable) Enable the <code>-Mi4</code> option
[NO]RECURSIVE	(Disable) Enable the <code>-Mrecursive</code> option

Option	Action Taken
[NO]REENTRANT	(Enable) Disable optimizations that may result in code that is not reentrant.
[NO]STANDARD	(Disable) Enable the <code>-Mstandard</code> option

The following restrictions apply to the `OPTIONS` statement:

- The `OPTIONS` statement must be the first statement in a program unit; it must precede the `PROGRAM`, `SUBROUTINE`, `FUNCTION`, and `BLOCKDATA` statements.
- The options listed in the `OPTIONS` statement override values from the driver command-line for the program unit (subprogram) immediately following the `OPTIONS` statement.
- Any abbreviated version of an option that is long enough to identify the option uniquely is a legal abbreviation for the option.
- Case is not significant, unless the `-Mupcase` is present on the command line. If it is, each option must be in lowercase.

## PARAMETER

The `PARAMETER` statement gives a symbolic name to a constant.

F77

### Syntax

```
PARAMETER ( name = expression [, name = expression... ] )
```

`expression`

is an arithmetic expression formed from constant or `PARAMETER` elements using the arithmetic operators `+`, `-`, `*`, `/`, `>`. The usual precedence order can be changed by using parentheses. The expression may include a previously defined `PARAMETER`.

### Examples

```
PARAMETER ( PI = 3.142 )
PARAMETER ( INDEX = 1024 )
PARAMETER ( INDEX3 = INDEX * 3 )
```

The following VAX/VMS extensions to the `PARAMETER` statement are fully supported:

- Its list is not bounded with parentheses.
- The form of the constant (rather than the implicit or explicit typing of the symbolic name) determines the data type of the variable.

The form of the alternative `PARAMETER` statement is:

```
PARAMETER p=c [,p=c]...
```

where `p` is a symbolic name and `c` is a constant, symbolic constant, or a compile time constant expression.

## PAUSE

(Obsolescent) The PAUSE statement stops the program's execution. The PAUSE statement is obsolescent because a WRITE statement may send a message to any device, and a READ statement may be used to wait for a message from the same device.

### Syntax

```
PAUSE [character-expression | digits ]
```

The PAUSE statement stops the program's execution. The program may be restarted later and execution will then continue with the statement following the PAUSE statement.

## POINTER

The POINTER statement provides a means for declaring pointers.

### F90

### Syntax

```
POINTER [::] object-name [ (deferred-shape-spec-list) ]  
[, object-name [ ( deferred-shape-spec-list ) ]]
```

### Example

```
REAL, DIMENSION(:, :), POINTER :: X
```

## POINTER (Cray)

The POINTER statement is an extension to FORTRAN 77. It declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its pointer-based variable.

### F77 extension

#### Syntax

```
POINTER (p1, v1) [, (p2, v2) ...]
```

#### v1 and v2

are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.

#### p1 and p2

are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced,



the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined. The pointer is defined in one of the following ways:

- By assigning the value of the LOC function.
- By assigning a value defined in terms of another pointer variable.
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

## Example

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0      ! IC gets 0
P = LOC(XC)
Q = P + 20 ! same as LOC(XC(6))
X(1) = 0   ! XC(6) gets 0
ALLOCATE (X) ! Q locates a dynamically
              ! allocated memory area
```

## Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is assumed not to overlap with another pointer-based variable.

## PRINT

The PRINT statement is a data transfer output statement.

### F77

### Syntax

```
PRINT format-specifier [, iolist]
```

or

```
PRINT namelist-group
```

format-specifier

a label of a format statement or a variable containing a format string or asterisk.

iolist

is an input/output list that must either be one of the items in an input list or any other expression. A character expression involving concatenation of an operand of variable length cannot be included in an output list, however, unless the operand is the symbolic name of a constant.

namelist-group

the name of the namelist group.

## Description

When a PRINT statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification.<sup>1</sup> The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h( )
print sum
```

and is roughly equivalent to

```
character*4 ch
ch = '( )'
print ch
```

## See Also

[“READ”](#), [“PRINT”](#)

## PRIVATE

The PRIVATE statement specifies entities defined in a module are not accessible outside of the module. This statement is only valid in a module. The default specification for a module is PUBLIC.

## F90

## Syntax

```
PRIVATE [ :: [ access-id-list ] ]
```

<sup>1</sup>If an asterisk (\*) is used instead of a format identifier, the list-directed formatting rules apply.

## Description

## Example

```
MODULE FORMULA
PRIVATE
PUBLIC :: VARA
.
.
.
END MODULE
```

## Type

Non-executable

## See Also

[“PUBLIC”](#), [“MODULE”](#)

## PROGRAM

The PROGRAM statement specifies the entry point for the linked Fortran program.

## F77

## Syntax

```
PROGRAM [name]
.
.
.
END [ PROGRAM [program-name]]
```

**name**

is optional; if supplied it becomes the name of the program module and must not clash with any other names used in the program. If it is not supplied, a default name is used.

## Description

The program statement specifies the entry point for the linked Fortran program. An END or END PROGRAM statement terminates the program.

The END PROGRAM statement terminates a main program unit that begins with the optional PROGRAM statement. The program name found in the END PROGRAM must match that in the PROGRAM statement.

## Example

```
PROGRAM MYOWN
REAL MEAN, TOTAL
.
CALL TRIG(A, B, C, MEAN)
.
END
```

## PUBLIC

The PUBLIC statement specifies entities defined in a module are accessible outside of the module. This statement is only valid in a module. The default specification for a module is PUBLIC.

F90

### Syntax

```
PUBLIC [ :: [ access-id-list ] ]
```

### Example

```
MODULE FORMULA
PRIVATE
PUBLIC :: VARA
.
.
.
END MODULE
```

### Type

Non-executable

### See Also

[“MODULE”](#), [“PRIVATE”](#)

## PURE

The PURE attribute indicates that a function or subroutine has no side effects. Use of PURE can enable additional opportunities for optimization, and for HPF indicates that a subroutine or function can be used in a FORALL statement or construct or within an INDEPENDENT DO loop.

F95

### Syntax

```
PURE [type-specification] FUNCTION
```

or

```
type-specification PURE FUNCTION
```

or

```
PURE SUBROUTINE
```

### Type

Non-executable

### See Also

[“FUNCTION”](#), [“SUBROUTINE”](#)

## READ

The READ statement is a data transfer input statement.

F90

### Syntax

```
READ ([unit=] u, format-specifier [,control-information] [iolist])
READ format-specifier [,iolist]
READ ([unit=] u, [NML=] namelist-group [,control-information])
```

where the `UNIT=` is optional and the external unit specifier `u` is an integer. `control-information` is an optional control specification that may contain one each of the following specifiers in any order.

#### Note

---

The specifier follows the unit number if the optional `UNIT` specifier keyword is not supplied.

**ADVANCE= spec**

`spec` is a character expression specifying the access method for file connection as either `YES` or `NO`.

**ASYNCHRONOUS= async**

`async` is a character expression specifying whether to allow the data transfer to be done asynchronously. The value specified may be `'YES'` or `'NO'`.

**BLANK= char\_expr**

`char_expr` is a scalar character expression that controls the default interpretation of blanks when using a format specification.

- When any trailing blanks are removed, `char_expr` is either `NULL` or `ZERO`.
- If `BLANK=` is specified, then a format or namelist-group name must also appear.
- If `BLANK=` is not specified and you specify `FORM='FORMATTED'`, `NULL` is the default.

**END=s**

`s` is an executable statement label for the statement used for processing an end of file condition.

**EOR=s**

`s` is an executable statement label for the statement used for processing an end of record condition.

**ERR=s**

`s` is an executable statement label for the statement used for processing an error condition.

**FMT= format**

`format` is a label of a format statement or a variable containing a format string.

**IOSTAT=ios**

`ios` is an integer variable or array element. `ios` is 0 if no error occurs, -1 if an EOF occurs, and a positive integer when there is an error.

**NML= namelist**

`namelist` is a namelist group

PAD=pad\_expr

pad\_expr is a scalar character expression that specifies if input records are padded with blanks. pad\_expr must evaluate to YES or NO.

- If the value is YES, a formatted input record is padded with blanks if an input list is specified and the format specification requires more data from a record than the record contains. YES is the default.
- If NO is specified, the input list and format specification must not require more characters from a record than the record contains.

### Note

---

The Pad= pad\_expr specifier is valid only on files being connected for formatted input/output, although it is ignored during output of a formatted record.

POS= pos\_expr

pos\_expr is a scalar integer expression that specifies the file position in a file storage unit. POS may also appear if the unit is connected for stream access.

If POS is present, REC can not be specified.

REC=rn

rn is a record number to read and must be a positive integer. This is only used for direct access files.

SIZE=n

n is the number of characters read.

iolist

(input list) must either be one of the items in an input list or any other expression.

## Description

When a READ statement is executed, the following operations are carried out:

- Data is transferred from the standard input device to the items specified in the input and format specification.<sup>2</sup>
- The data are transferred between the specified destinations in the order specified by the input/output list.

Every item whose value is to be transferred must be defined.

## Example

```
READ(2,110) I,J,K
110 FORMAT(I2, I4, I3)
```

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h(i)
```

```
read sum, j
```

and is roughly equivalent to

```
character*4 ch
ch = '(i)'
read ch, j
```

## See Also

[“OPEN”](#), [“PRINT”](#), [“WRITE”](#)

## REAL

The REAL statement establishes the data type of a variable by explicitly attaching the name of a variable to a data type. This overrides the data typing implied by the initial letter of a symbolic name.

## F90

### Syntax

The syntax for REAL has two forms, a standard Fortran 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
REAL [ ( [ KIND = ] kind-value)
      [, attribute-list ::] entity-list
```

REAL permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

**attribute-list**

is the list of attributes for the character variable.

**entity-list**

is the list of defined entities.

### Syntax Extension

```
REAL [*n] name [*n] [dimensions] [/clist/]
      [,name] [*n] [dimensions][/clist/]...
```

**n**

is an optional size specification.

**name**

is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement below for an explanation of array declarators).

**clist**

is a list of constants that initialize the data, as in a DATA statement.

### Description

The REAL type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled.

## Note

---

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, **IMPLICIT** statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An **IMPLICIT** statement overrides predefined typing rules.

The default size of a **REAL** variable is 4 bytes. With the **-Mr8** option, the default size of an **REAL** variable is 8 bytes.

## Example

```
REAL KNOTS
```

## RECORD

The **RECORD** statement, a VAX Fortran extension, defines a user-defined aggregate data item.

## F77 extension

## Syntax

```
RECORD /structure_name/record_namelist
  [,/structure_name/record_namelist]
  ...
  [,/structure_name/record_namelist]
END RECORD
```

**structure\_name**

is the name of a previously declared structure.

**record\_namelist**

is a list of one or more variable or array names separated by commas.

## Description

You create memory storage for a record by specifying a structure name in the **RECORD** statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (**.**), and the field name (for example, *recordname.fieldname*). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, **INTEGER**), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of the **COMMON**, **SAVE**, **NAMELIST**, **DATA** and **EQUIVALENCE** statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

Records are allowed in **COMMON** and **DIMENSION** statements.

## Example

```
STRUCTURE /PERSON/ ! Declare a structure
```



```

defining a person
  INTEGER ID
  LOGICAL LIVING
  CHARACTER*5 FIRST, LAST, MIDDLE
  INTEGER AGE
END STRUCTURE
! Define population to be an array where each element is of
! type person. Also define a variable, me, of type person.
RECORD /PERSON/ POPULATION(2), ME
...
ME.AGE = 34          ! Assign values for the variable me
ME.LIVING = .TRUE. ! to some of the fields.
ME.FIRST = 'Steve'
ME.ID = 542124822
...
POPULATION(1).LAST = 'Jones' ! Assign the "LAST" field of
                             ! element 1 of array population.
POPULATION(2) = ME          ! Assign all the values of record
                             ! "ME" to the record population(2)

```

## RECURSIVE

The **RECURSIVE** statement indicates whether a function or subroutine may call itself recursively.

F90

Syntax

```
RECURSIVE [type-specification] FUNCTION
```

or

```
type-specification RECURSIVE FUNCTION
```

or

```
RECURSIVE SUBROUTINE
```

Type

Non-executable

See Also

[“FUNCTION”](#), [“SUBROUTINE”](#)

## REDIMENSION

The **REDIMENSION** statement, a CRAY extension to FORTRAN 77, dynamically defines the bounds of a deferred-shape array. After a **REDIMENSION** statement, the bounds of the array become those supplied in the statement, until another such statement is encountered.

F77 extension

Syntax

```
REDIMENSION name ([lb:]ub[, [lb:]ub]...)
```

## RETURN

```
[ ,name( [lb:]ub[ , [lb:]ub]... ) ]...
```

Where:

**name**

is the symbolic name of an array.

**[lb:]ub**

is a dimension declarator specifying the bounds for a dimension (the lower bound lb and the upper bound ub). lb and ub must be integers with ub greater than lb. The lower bound lb is optional; if it is not specified, it is assumed to be 1. The number of dimension declarations must be the same as the number of dimensions in the array.

### Example

```
REAL A( : , : )
POINTER (P, A)
P = malloc(12 * 10 * 4)
REDIMENSION A(12, 10)
A(3, 4) = 33.
```

## RETURN

The RETURN statement causes a return to the statement following a CALL when used in a subroutine, and returns to the relevant arithmetic expression when used in a function.

F77

### Syntax

```
RETURN
```

### Alternate RETURN

(Obsolescent) The alternate RETURN statement is obsolescent for HPF and Fortran 90/95. Use the CASE statement where possible in new or updated code. The alternate RETURN statement takes the following form:

```
RETURN expression
```

**expression**

expression is converted to integer if necessary (expression may be of type integer or real). If the value of expression is greater than or equal to 1 and less than or equal to the number of asterisks in the SUBROUTINE or subroutine ENTRY statement then the value of expression identifies the nth asterisk in the actual argument list and control is returned to that statement.

### Example

```
SUBROUTINE FIX (A,B,*,*,C)
40 IF (T) 50, 60, 70
50 RETURN
60 RETURN 1
70 RETURN 2
END
```

```

PROGRAM FIXIT
CALL FIX(X, Y, *100, *200, S)
WRITE(*,5) X, S ! Come here if (T) < 0
STOP
100 WRITE(*, 10) X, Y ! Come here if (T) = 0
STOP
200 WRITE(*,20) Y, S ! Come here if (T) > 0

```

## REWIND

The REWIND statement positions the file at its beginning. The statement has no effect if the file is already positioned at the start or if the file is connected but does not exist.

F77

### Syntax

```

REWIND unit
REWIND (unit,list)

```

**unit**

is an integer value which is the external unit.

**list**

contains the optional specifiers as follows:

**UNIT=unit**

unit is the unit specifier.

**ERR=errs**

errs is an executable statement label for the statement used for processing an error condition. If an error condition occurs execution continues with the statement specified by errs.

**IOSTAT=ios**

ios is an integer value or array element. ios is 0 if no error occurs and a positive integer when there is an error.

### Examples

```

REWIND 5
REWIND(2, ERR=30)
REWIND(3, IOSTAT=IOERR)

```

## SAVE

The SAVE statement retains the definition status of an entity after a RETURN or END statement in a subroutine or function has been executed.

F77

### Syntax

```

SAVE [v [, v ]...]

```

## SELECT CASE

v

name of array, variable, or common block (enclosed in slashes)

### Description

Using a common-block name, preceded and followed by a slash, ensures that all entities within that COMMON block are saved.

- SAVE may be used without a list, in which case all the allowable entities within the program unit are saved. This approach is the same as using the `-Msave` command-line option.
- Dummy arguments, names of procedures and names of entities within a common block may not be specified in a SAVE statement.
- Use of the SAVE statement with local variables ensures the values of the local variables are retained for the next invocation of the SUBROUTINE or FUNCTION.
- Within a main program the SAVE statement is optional and has no effect.

When a RETURN or END is executed within a subroutine or function, all entities become undefined with the exception of:

- Entities specified by a SAVE statement
- Entities in blank common or named common
- Entities initially defined which have not been changed in any way

### Example

```
PROGRAM SAFE
...
CALL KEEP
...
SUBROUTINE KEEP
COMMON /LIST/ TOP, MIDDLE
INTEGER LOCAL1
...
SAVE /LIST/, LOCAL1
```

## SELECT CASE

The SELECT CASE statement begins a CASE construct.

F90

### Syntax

```
[case-name:]SELECT CASE (case-expr)
[ CASE selector [name]
block]
...
[ CASE DEFAULT [case-name]
block]
```

```
END SELECT [case-name]
```

## Description

The SELECT CASE statement defines the case expression that is to be evaluated.

## Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
  TYPE=1
CASE ( 4:6 )
  TYPE=2
CASE DEFAULT
  TYPE=0
END SELECT
```

## SEQUENCE

The SEQUENCE statement is a derived type qualifier that specifies the ordering of the storage associated with the derived type. This statement specifies storage for use with COMMON and EQUIVALENCE statements (the preferred method for derived type data sharing is using MODULES).

### Note

---

There is also an HPF SEQUENCE directive that specifies whether an array, common block, or equivalence is sequential or non-sequential. Refer to the PGHPF User's Guide for more information.

## F90

### Syntax

```
TYPE
  [SEQUENCE]
  type-specification...
END TYPE
```

### Example

```
TYPE RECORD
  SEQUENCE
  CHARACTER NAME(25)
  INTEGER CUST_NUM
  REAL COST
END TYPE
```

## STOP

The STOP statement stops the program's execution and precludes any further execution of the program.

## F77

### Syntax

```
STOP [character-expression | digits ]
```

## STRUCTURE

The STRUCTURE statement, a VAX extension to FORTRAN 77, defines an aggregate data type.

### F77 VAX extension

#### Syntax

```
STRUCTURE [/structure_name/][field_namelist]
  field_declaration
  [field_declaration]
  ...
  [field_declaration]
END STRUCTURE
```

#### structure\_name

is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

#### 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

can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

#### Description

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications. Note that aligning of structure fields is not supported by VAX/VMS Fortran.

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. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware and a structure's storage requirements are therefore machine-dependent. Note that VAX/VMS Fortran does no padding. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

Data initialization can occur for the individual fields.

The UNION and MAP statements are supported.

The following is an example of record and structure usage.

```
STRUCTURE /account/
  INTEGER typetag  ! Tag to determine defined map
  UNION
  MAP              ! Structure for an employee
  CHARACTER*12 ssn ! Social Security Number
  REAL*4 salary
```

```

CHARACTER*8 empdate ! Employment date
END MAP
MAP                ! Structure for a customer
INTEGER*4 acct_cust
REAL*4 credit_amt
CHARACTER*8 due_date
END MAP
MAP                ! Structure for a supplier
INTEGER*4 acct_supp
REAL*4 debit_amt
BYTE num_items
BYTE items(12)    ! Items supplied
END MAP
END UNION
END STRUCTURE
RECORD /account/ recarr(1000)

```

## SUBROUTINE

The **SUBROUTINE** statement introduces a subprogram unit. The statements that follow should be laid out in the same order as a **PROGRAM** module.

F77

### Syntax

```

[RECURSIVE] SUBROUTINE name &
[(argument[,argument...])] &
[specification-part]
[execution-part]
[internal-subspart]
END [SUBROUTINE [name]]

```

#### name

is the name of the subroutine being declared and must be unique among all the subroutine and function names in the program. name should not clash with any local, **COMMON**, **PARAMETER** or **ENTRY** names.

#### argument

is a symbolic name, starting with a letter and containing only letters and digits. The type of argument can be **REAL**, **INTEGER**, **DOUBLE PRECISION**, **CHARACTER**, **COMPLEX**, or **BYTE**, etc.

#### specification-part

is the specification of data types for the subroutine.

#### execution-part

contains the subprogram's executable statements.

#### internal-subspart

contains subprograms defined within the subroutine.

### Description

A **SUBROUTINE** must be terminated by an **END** statement. The statements and names in the subprogram only apply to the subroutine except for subroutine or function references and the names of **COMMON** blocks. Dummy arguments may be specified as **\*** which indicates that the **SUBROUTINE** contains alternate returns.

## TARGET

Recursion is allowed if the `-Mrecursive` option is used on the command-line or the `RECURSIVE` prefix is included in the function definition.

### Example

```
SUBROUTINE DAXTIM (A, X, Y, N, M, ITER, FP, TOH)
  INTEGER*4 N, M, ITER
  REAL*8 A, X(N,M), Y(N,M), FP, TOH
  .
  .
END SUBROUTINE DAXTIM
```

### See Also

[“PURE”](#), [“RECURSIVE”](#)

## TARGET

The `TARGET` specification statement (attribute) specifies that a data type may be the object of a pointer variable (e.g., pointed to by a pointer variable). Likewise, types that do not have the `TARGET` attribute cannot be the target of a pointer variable.

### F90

### Syntax

```
TARGET [ :: ] object-name [(array-spec)]
[, object-name [(array-spec)]]...
```

### See Also

[“ALLOCATABLE”](#), [“POINTER”](#)

## THEN

The `THEN` statement is part of a block `IF` statement and surrounds a series of statements that are conditionally executed.

### F77

### Syntax

```
IF logical expression THEN
  statements
ELSE IF logical expression THEN
  statements
ELSE
  statements
ENDIF
```

The `ELSE IF` section is optional and may be repeated any number of times. Other `IF` blocks may be nested within the statements section of an `IF` block.



## Example

```

IF ( I.GT.70 ) THEN
  M=1
ELSE IF ( I.LT.5 ) THEN
  M=2
ELSE IF ( I.LT.16 ) THEN
  M=3
ENDIF
IF ( I.LT.15 ) THEN
  M = 4
ELSE
  M=5
ENDIF

```

## TYPE

The **TYPE** statement begins a derived type data specification or declares variables of a specified user-defined type.

### F90

#### Syntax for Type Declaration

```
TYPE (type-name) [ , attribute-list :: ] entity-list
```

#### Syntax for Derived Type Definition

```

TYPE [[ access-spec ] :: ] type-name
  [ private-sequence-statement ] ...
  component-definition-statement
  [ component-definition-statement ]...
END TYPE [type-name]

```

#### Syntax for FORTRAN 77 Type Statement

```
TYPE
```

The **TYPE** statement has the same syntax and effect as the **PRINT** statement. For a full description, refer to [“PRINT,” on page 103](#).

## UNION

A **UNION** declaration, a DEC extension to FORTRAN 77, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a UNION statement is as follows:

## F77 extension

### Syntax

```
UNION
  map_declaration
  [map_declaration]
  ...
  [map_declaration]
END UNION
```

The format of the map\_declaration is as follows:

```
MAP
  field_declaration
  [field_declaration]
  ...
  [field_declaration]
END MAP
```

#### field\_declaration

where 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 typed data field within a union.

### Description

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area 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 field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
  INTEGER typetag      ! Tag to determine defined map.
  UNION
  MAP                  ! Structure for an employee
  CHARACTER*12 ssn     ! Social Security Number
  REAL*4 salary
  CHARACTER*8 empdate  ! Employment date
  END MAP
  MAP                  ! Structure for a customer
```

```

INTEGER*4 acct_cust
REAL*4 credit_amt
CHARACTER*8 due_date
END MAP
MAP          ! Structure for a supplier
INTEGER*4 acct_supp
REAL*4 debit_amt
BYTE num_items
BYTE items(12)      ! Items supplied
END MAP
END UNION
END STRUCTURE
RECORD /account/ recarr(1000)

```

## USE

The USE statement gives a program unit access to the public entities or to the named entities in the specified module.

## F90

### Syntax

```

USE module-name [, rename-list ]
USE module-name, ONLY: [ only-list ]

```

**module-name**

is the name of a file that has an associated compiled .mod file which is included when the module is used.

**rename-list**

is a comma-separated list of rename.

**ONLY: [only-list]**

is the method to restrict access, limiting it to only those entities specified in only-list.

### Description

The USE statement makes modules available. A module-name file has an associated compiled .mod file that is included when the module is used. The .mod file is searched for in the following directories:

- Each `-I` directory specified on the command-line.
- The directory containing the file that contains the USE statement (the current working directory.)
- The standard include area.

A module's public specifications include declared variables, named constants, derived-type definitions, procedure interfaces, procedures, generic identifiers, and namelist groups.

All USE statements must appear after the program unit header statement and before any other statements. More than one USE statement may be present, including more than one referring to the same module.

Modules may contain USE statements referring to other modules; however, references must not directly or indirectly be recursive. The USE statement makes available by use association all publicly accessible entities in the module, except that the USE statement may rename some module entities.

## VOLATILE

The same name or specifier may be made accessible by means of two or more USE statements. Such an entity must not be referenced in the scoping unit containing the USE statements, except where specific procedures can be distinguished by the overload rules. A rename or ONLY clause may be used to restrict access to one name or to rename one entity so that both are accessible.

### Examples

```
USE MOD1
USE MOD2, TEMP => VAR
USE MOD3, ONLY: RESULTS, SCORES => VAR2
```

### Type

Non-executable

### See Also

[“MODULE”](#)

## VOLATILE

The VOLATILE statement inhibits all optimizations on the variables, arrays and common blocks that it identifies. The VOLATILE attribute, added in Fortran 2003, is used in a type declaration statement.

### F77 extension

### Syntax

### Volatile Attribute

```
datatype, volatile :: var_name
OR
datatype :: var_name
volatile :: var_name
var_name
```

### Volatile Statement

```
VOLATILE nitem [, nitem ...]
```

nitem

is the name of a variable, an array, or a common block enclosed in slashes.

### Description

Being volatile indicates to the compiler that, at any time, the variable might change or be examined from outside the Fortran program. The impact on the programmer is that anytime a volatile variable is referenced, the value must be loaded from memory. Furthermore, any assignment to the volatile variable must be written to memory.

If *nitem* names a common block, all members of the block are volatile. The volatile attribute of a variable is inherited by any direct or indirect equivalences, as shown in the example.

### Volatile Attribute Example

The following example declares both the integer variable `xyz` and the real variable `abc` to be volatile.

```
integer, volatile :: xyz
real :: abc
volatile :: abc
```

### Volatile Statement Example

```
COMMON /COM/ C1, C2
VOLATILE /COM/, DIR      ! /COM/ and DIR are volatile
EQUIVALENCE (DIR, X)    ! X is volatile
EQUIVALENCE (X, Y )    ! Y is volatile
```

## WAIT

Performs a wait operation for specified pending asynchronous data transfer operations.

### F2003

#### Syntax

```
WAIT (wait_specs_list)
```

`wait_specs_list` can include any of the following specifiers:

**UNIT = ] file-unit-number**

A file-unit-number must be specified. If the optional characters `UNIT=` are omitted, the file-unit-number is the first item in the wait-spec-list.

**END = label**

`label` must be the statement label of a branch target statement that appears in the same scoping unit as the `WAIT` statement.

`END=` specifier has no effect if the pending data transfer operation is not a `READ`.

**EOR = label**

`label` must be the statement label of a branch target statement that appears in the same scoping unit as the `WAIT` statement.

`EOR=` specifier has no effect if the pending data transfer operation is not a nonadvancing `READ`.

**ERR = label**

`label` must be the statement label of a branch target statement that appears in the same scoping unit as the `WAIT` statement.

**ID = scalar\_int\_var**

`scalar_int_var` is the identifier of a pending data transfer operation for the specified unit.

- If the `ID=` specifier appears, a wait operation for the specified data transfer operation is performed.

## WHERE

- If the ID= specifier is omitted, wait operations for all pending data transfers for the specified unit are performed.

IOMSG = iomsg-var

iomsg-var is an I/O message variable.

IOSTAT =scalar-int-var

scalar\_int\_var is the identifier of a pending data transfer operation for the specified unit.

For more information on IOSTAT, ERR=, EOR=, END=, and IOMSG=, refer to the READ and WRITE statements.

## Description

This statement performs a wait operation for specified pending asynchronous data transfer operations.

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or that was not opened for asynchronous input/output is permitted, provided that the WAIT statement has no ID= specifier. This type of WAIT statement does not cause an error or end-of-file condition to occur.

### Note

---

No specifier shall appear more than once in a given wait-spec-list.

## Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 )
  GRADE = 'P'
ELSE WHERE
  GRADE = 'F'
END WHERE
```

## WHERE

The WHERE statement and the WHERE END WHERE construct permit masked assignments to the elements of an array (or to a scalar, zero dimensional array).

## F90

## Syntax

### WHERE Statement

```
WHERE (logical-array-expr) array-variable = array-expr
```

### WHERE Construct

```
WHERE (logical-array-expr)
  array-assignments
```

```
[ ELSE WHERE
  array-assignments ]
END WHERE
```

## Description

This construct allows for conditional assignment to an array based on the result of a logical array expression. The logical array expression and the array assignments must involve arrays of the same shape.

## Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 )
  GRADE = 'P'
ELSE WHERE
  GRADE = 'F'
END WHERE
```

## WRITE

The WRITE statement is a data transfer output statement.

## F90

## Syntax

```
WRITE ([unit=] u, [,control-information]) [iolist]
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

where the UNIT= is optional and the external unit specifier u is an integer. This may also be a \* indicating output to the default output unit, 6.

In addition to the unit specification, control-information are optional control specifications, and may be any of those listed in the following (there are some limits on the allowed specifications depending on the type of output, for example, non-advancing, direct and sequential):

**ADVANCE=spec**

spec is a character expression specifying the access method for the write. YES indicates advancing formatted sequential data transfer. NO indicates non-advancing formatted sequential data transfer.

**ASYNCHRONOUS= async**

async is a character expression specifying whether to allow the data transfer to be done asynchronously. The value specified may be 'YES' or 'NO'.

**DELIM=char\_expr**

char\_expr specifies the delimiter, if any, that delimits character constants written with list-directed or namelist formatting. char\_expr is a scalar character expression whose value must be one of the following:

- APOSTROPHE - Apostrophes delimit character constants and all apostrophes within character constants are doubled.
- QUOTE - Double quotation marks delimit character constants and all double quotation marks within character constants are doubled.

- NONE. Character constants are not delimited and no characters are doubled.

The default value is NONE.

The DELIM = specifier is valid only for files being connected for formatted input/output, although it is ignored during input of a formatted record.

ERR=s

s is an executable statement label for the statement used for processing an error condition.

[FMT=]format

format is a label of a format statement or a character expression that contains a format string.

IOSTAT=ios

ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

[NML=] namelist

namelist is the name of a namelist group

POS= pos\_expr

pos\_expr is a scalar integer expression that specifies the file position in a file storage unit. POS may also appear if the unit is connected for stream access.

If POS is present, REC can not be specified.

REC=rn

rn is a record number to read and must be a positive integer. This is only used for direct access files.

iolist

iolist must either be one of the items in an input list or any other expression. However a character expression involving concatenation of an operand of variable length cannot be included in an output list unless the operand is the symbolic name of a constant.

## Description

When a WRITE statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification.<sup>3</sup> The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

## Example

```
WRITE (6,90) NPAGE
90 FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

## Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

<sup>3</sup>If an asterisk (\*) is used instead of a format identifier, the list-directed formatting rules apply.



```
real sum
sum = 4h(a)
write(*,sum) string
```

and is roughly equivalent to

```
character*4 ch
ch = '(a)'
write(*,ch) string
```

See Also

[“READ”](#), [“PRINT”](#)



# Chapter 4. Fortran Arrays

Fortran arrays are any object with the dimension attribute. In Fortran 90/95, and in HPF, arrays may be very different from arrays in older versions of Fortran. Arrays can have values assigned as a whole without specifying operations on individual array elements, and array sections can be accessed. Also, allocatable arrays that are created dynamically are available as part of the Fortran 90/95 and HPF standards. Arrays in HPF play a central role in data distribution and data alignment (refer to this chapter and The High Performance Fortran Handbook for details on working with arrays in HPF). This chapter describes some of the features of Fortran 90/95 and HPF arrays.

The following example illustrates valid array operations.

```
REAL(10,10) A,B,C
A=12 !Assign 12 to all elements of A
B=3 !Assign 3 to all elements of B
C=A+B !Add each element of A to each of B
```

## Array Types

Fortran supports four types of arrays: explicit-shape arrays, assumed-shape arrays, deferred-shape arrays and assumed-size arrays. Both explicit-shape arrays and deferred shape arrays are valid in a main program. Assumed shape arrays and assumed size arrays are only valid for arrays used as dummy arguments. Deferred shape arrays, where the storage for the array is allocated during execution, must be declared with either the `ALLOCATABLE` or `POINTER` attributes.

Every array has properties of type rank, shape and size. The extent of an array's dimension is the number of elements in the dimension. The array rank is the number of dimensions in the array, up to a maximum of seven. The shape is the vector representing the extents for all dimensions. The size is the product of the extents. For some types of arrays, all of these properties are determined when the array is declared. For other types of arrays, some of these properties are determined when the array is allocated or when a procedure using the array is entered. For arrays that are dummy arguments, there are several special cases.

Allocatable arrays are arrays that are declared but for which no storage is allocated until an `allocate` statement is executed when the program is running. Allocatable arrays provide Fortran 90/95 and HPF programs with dynamic storage. Allocatable arrays are declared with a rank specified with the ":" character rather than with explicit extents, and they are given the `ALLOCATABLE` attribute.

## Explicit Shape Arrays

Explicit shape arrays are those arrays familiar to FORTRAN 77 programmers. Each dimension is declared with an explicit value. There are two special cases of explicit arrays. In a procedure, an explicit array whose bounds are passed in from the calling program is called an automatic-array. The second special case, also found in a procedure, is that of an adjustable-array which is a dummy array where the bounds are passed from the calling program.

## Assumed Shape Arrays

An assumed shape array is a dummy array whose bounds are determined from the actual array. Intrinsic called from the called program can determine sizes of the extents in the called program's dummy array.

## Deferred Shape Arrays

A deferred shape array is an array that is declared, but not with an explicit shape. Upon declaration, the array's type, its kind, and its rank (number of dimensions) are determined. Deferred shape arrays are of two varieties, allocatable arrays and array pointers.

## Assumed Size Arrays

An assumed size array is a dummy array whose size is determined from the corresponding array in the calling program. The array's rank and extents may not be declared the same as the original array, but its total size (number of elements) is the same as the actual array. This form of array should not need to be used in new Fortran programs.

## Array Specification

Arrays may be specified in either of two types of data type specification statements, attribute-oriented specifications or entity-oriented specifications. Arrays may also optionally have data assigned to them when they are declared. This section covers the basic form of entity-based declarations for the various types of arrays. Note that all the details of array passing for procedures are not covered here; refer to The Fortran 95 Handbook for complete details on the use of arrays as dummy arguments.

## Explicit Shape Arrays

Explicit shape arrays are defined with a specified rank, each dimension must have an upper bound specified, and a lower bound may be specified. Each bound is explicitly defined with a specification of the form:

```
[lower-bound:] upper-bound
```

An array has a maximum of seven dimensions. The following are valid explicit array declarations:

```
INTEGER NUM1(1,2,3)           ! Three dimensions
INTEGER NUM2(-12:6,100:1000) ! Two dimensions with lower & upper bounds
INTEGER NUM3(0,12,12,12)     ! Array of size 0
INTEGER NUM3(M:N,P:Q,L,99)   ! Array with 4 dimensions
```

## Assumed Shape Arrays

An assumed shape array is always a dummy argument. An assumed shape array has a specification of the form:

```
[lower-bound] :
```

The number of colons (:) determines the array's rank. An assumed shape array cannot be an ALLOCATABLE or POINTER array.

## Deferred Shape Arrays

An deferred shape array is an array pointer or an allocatable array. An assumed shape array has a specification that determines the array's rank and has the following form for each dimension:

```
:
```

For example:

```
INTEGER, POINTER :: NUM1 (:, :, :, : )
INTEGER, ALLOCATABLE :: NUM2 ( : )
```

## Assumed Size Arrays

An assumed size array is a dummy argument with an assumed size. The array's rank and bounds are specified with a declaration that has the following form:

```
[explicit-shape-spec-list , ][lower-bound:]*
```

For example:

```
SUBROUTINE YSUM1 (M, B, C)
  INTEGER M
  REAL, DIMENSION (M, 4, 5, *) :: B, C
```

## Array Subscripts and Access

There are a variety of ways to access an array in whole or in part. Arrays can be accessed, used, and assigned to as whole arrays, as elements, or as sections. Array elements are the basic access method.

In the following example, the value of 5 is assigned to element 3,1 of NUMB.

```
INTEGER, DIMENSION (3, 11) :: NUMB
NUMB (3, 1) = 5
```

The following statement assigns the value 5 to all elements of NUMB.

The array NUMB may also be accessed as an entire array:

```
NUMB = 5
```

## Array Sections and Subscript Triplets

Another possibility for accessing array elements is the array section. An array section is an array accessed by a subscript that represents a subset of the entire array's elements and is not an array element. An array section resulting from applying a subscript list may have a different rank than the original array. An array section's subscript list consists of subscripts, subscript triplets, and/or vector subscripts.

The following example uses a subscript triplet and a subscript, assigning the value 6 to all elements of NUMB with the second dimension of value 3 (NUMB(1,3), NUMB(2,3), NUMB(3,3) ).

```
NUMB ( : , 3 ) = 6
```

The following array section uses the array subscript triplet and a subscript to access three elements of the original array. This array section could also be assigned to a rank one array with three elements, as shown here:

```
INTEGER(3,11) NUMB
INTEGER(3) NUMC
NUMB(:,3)=6
NUMC=NUMB(:,3)
```

In this example, NUMC is rank 1 and NUMB is rank 2. This assignment, using the subscript 3, illustrates how NUMC, and the array section of NUMB, has a shape that is of a different rank than the original array.

The general form for an array's dimension with a vector subscript triplet is:

```
[subscript] : [subscript] [:stride]
```

The first subscript is the lower bound for the array section, the second is the upper bound and the third is the stride. The stride is by default one. If all values except the : are omitted, then all the values for the specified dimensions are included in the array section.

In the following example, using the NUMB previously defined, the statement has a stride of 2, and assigns the value 7 to the elements NUMB(1,3) and NUMB(3,3).

```
NUMB(1:3:2,3)=7
```

## Array Sections and Vector Subscripts

Vector-valued subscripts specify an array section by supplying a set of values defined in a one dimensional array (vector) for a dimension or several dimensions of an array section.

In the following example, the array section uses the vectors I and J to assign the value 7 to each of the elements: NUMB(2,1), NUMB(2,2), NUMB(3,1), and NUMB(3,2).

```
INTEGER J(2), I(2)
INTEGER NUMB(3,6)
I=(/1,2/)
J=(/2,3/)
NUMB(J,I)=7
```

## Array Constructors

An array constructor can be used to assign values to an array. Array constructors form one-dimensional vectors to supply values to a one-dimensional array, or one dimensional vectors and the RESHAPE function to supply values to arrays with more than one dimension.

Array constructors can use a form of implied DO similar to that in a DATA statement. For example:

```
INTEGER DIMENSION(4):: K = (/1,2,7,11/)
INTEGER DIMENSION(20):: J = (/ (I,I=1,40,2) /)
```

## CM Fortran Extensions

### The ARRAY Attribute §

The PGHPF compiler provides several extensions for handling arrays. The compiler handles the CM Fortran attribute ARRAY. The ARRAY attribute is similar to the DIMENSION attribute. Refer to “[ARRAY,” on page 44](#) for more details on the ARRAY statement.

## Array Constructors Extensions §

The PGHPF compiler supports an extended form of the array constructor specification. In addition to the (/ ../) specification for array constructors, PGHPF supports the notation where [ and ] begin and end, respectively, an array constructor.

In addition, an array constructor item may be a 'subscript triplet' in the form of an array section where the values are assigned to the array:

```
lower-bound : upper-bound [ : <stride> ]
```

For the values  $i : j : k$  the array would be assigned values  $i, i+k, i+2k, \dots, j$ . If  $k$  is not present, stride is assumed to be 1.

For example:

```
INTEGER, DIMENSION(20):: K = [1:40:2]
```





# Chapter 5. Input and Output

Input, output, and format statements provide the means for transferring data to or from files. Data is transferred as records to or from files. A record is a sequence of data which may be values or characters and a file is a sequence of such records. A file may be internal, that is, held in memory, or external such as those held on disk. To access an external file a formal connection must be made between a unit, for example a disk file, and the required file. An external unit must be identified either by a positive integer expression, the value of which indicates a unit, or by an asterisk (\*) which identifies a standard input or output device.

This chapter describes the types of input and output available and provides examples of input, output and format statements. There are four types of input/output used to transfer data to or from files: unformatted, formatted, list directed, and namelist.

- unformatted data is transferred between the item(s) in the input/output list (iolist) and the current record in the file. Exactly one record may be read or written.
- formatted data is edited to conform to a format specification, and the edited data is transferred between the item or items in the iolist, and the file. One or more records may be read or written. Non-advancing formatted data transfers are a variety of formatted I/O where a portion of a data record is transferred with each input/output statement.
- list directed input/output is an abbreviated form of formatted input/output that does not use a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to or from the file, using a default, and not necessarily accurate format specification.
- namelist input/output is a special type of formatted data transfer; data is transferred between a named group (namelist group) of data items and one or more records in a file.

## File Access Methods

You can access files using one of two methods, sequential access, or direct access (random access). The access method is determined by the specifiers supplied when the file is opened using the OPEN statement. Sequential access files are accessed one after the other, and are written in the same manner. Direct access files are accessed by specifying a record number for input, and by writing to the currently specified record on output.

Files may contain one of two types of records, fixed length records or variable length records. To specify the size of the fixed length records in a file, use the RECL specifier with the OPEN statement. RECL sets the record length in bytes.<sup>1</sup> RECL can only be used when access is direct.

A record in a variable length formatted file is terminated with \n. A record in a variable length unformatted file is preceded and followed by a word indicating the length of the record.

### Standard Preconnected Units

Certain input and output units are predefined, depending on the value of compiler options. The PGI Fortran compilers `-Mdefaultunit` option tells the compiler to treat "\*" as a synonym for standard input for reading and standard output for writing. When the option is `-Mnodefaultunit`, the compiler treats "\*" as a synonym for unit 5 on input and unit 6 on output.

## Opening and Closing Files

The OPEN statement establishes a connection to a file. OPEN allows you to do any of the following

- Connect an existing file to a unit.
- Create and connect a file to a unit.
- Create a file that is preconnected.
- Establish the access method and record format for a connection.

OPEN has the form:

```
OPEN (list)
```

where list contains a unit specifier of the form:

```
[UNIT=] u
```

where u, an integer, is the external unit specifier.

In addition list may contain one of each of the specifiers shown in [Table 5.1, "OPEN Specifiers"](#).

### Direct Access Files

If a file is connected for direct access using OPEN with ACCESS='DIRECT', the record length must be specified using RECL=, and optionally one of each of the other specifiers may be used.

Any file opened for direct access must be via fixed length records.

In the following example:

- A new file, book.dat, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters.

---

<sup>1</sup>The units depend on the value of the FORTRANOPT environment variable. If the value is vaxio, then the record length is in units of 32-bit words. If FORTRANOPT is not defined, or its value is something other than vaxio, then the record length is always in units of bytes.

- Blank values are ignored in numeric values.
- If an error condition exists when the OPEN statement is executed, the variable E1 is assigned some positive value, and then execution continues with the statement labeled 20.
- If no error condition pertains, E1 is assigned the value 0 and execution continues with the statement following the OPEN statement.

```
OPEN( 12, IOSTAT=E1, ERR=20, FILE= 'book.dat', BLANK= 'NULL',
+ACCESS= 'DIRECT', RECL=98, FORM= 'FORMATTED', STATUS= 'NEW' )
```

## Closing a File

Close a unit by specifying the CLOSE statement from within any program unit. If the unit specified does not exist or has no file connected to it, the CLOSE statement has no effect.

Provided the file is still in existence, it may be reconnected to the same or a different unit after the execution of a CLOSE statement. An implicit CLOSE is executed when a program stops.

The CLOSE statement terminates the connection of the specified file to a unit.

```
CLOSE ([UNIT=] u [, IOSTAT=ios] [, ERR= errs ]
[, STATUS= sta] [, DISPOSE= sta] [, DISP= sta])
```

CLOSE takes the status values IOSTAT, ERR, and STATUS, similar to those described in [Table 5.1, “OPEN Specifiers”](#). In addition, CLOSE allows the DISPOSE or DISP specifier which can take a status value sta which is a character string, where case is insignificant, specifying the file status (the same keywords are used for the DISP and DISPOSE status). Status can be KEEP or DELETE. KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement, conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

Table 5.1. OPEN Specifiers

Specifier	Description
ACCESS=acc	Where acc is a character string specifying the access method for file connection as DIRECT (random access) or SEQUENTIAL. The default is SEQUENTIAL.
ACTION=act	Where act is a character string specifying the allowed actions for the file and is one of READ, WRITE, or READWRITE.
BLANK=blnk	Where blnk is a character string which takes the value NULL or ZERO: NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. The default is NULL. This specifier must only be used when a file is connected for formatted input/output.
DELIM=del	Specify the delimiter for character constants written by a list-directed or namelist-formatted statement. The options are APOSTROPHE, QUOTE, and NONE.

Specifier	Description
ERR=errs	An error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs, execution continues with the statement specified by errs. <sup>2</sup>
FILE=fin	Where fin is a character string defining the file name to be connected to the specified unit.
FORM=fm	Where fm is a character string specifying whether the file is being connected for FORMATTED or UNFORMATTED output respectively. The default is FORMATTED.
IOSTAT=ios	Input/output status specifier where ios is an integer scalar memory reference. If this is included in list, ios becomes defined with 0 if no error exists or a positive integer when there is an error condition. <sup>a</sup>
PAD=padding	Specifies whether or not to use blank padding for input items. The padding values are YES and NO. The value NO requires that the input record and the input list format specification match.
POSITION=pos	Specifies the position of an opened file. ASIS indicates the file position remains unchanged. REWIND indicates the file is to be rewound, and APPEND indicates the file is to be positioned just before an end-of-file record, or at its terminal point.
RECL=rl	Where rl is an integer which defines the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.
STATUS=sta	The file status where sta is a character expression: it can be NEW, OLD, SCRATCH, REPLACE or UNKNOWN. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default is UNKNOWN.

<sup>a</sup>If IOSTAT and ERR are not present, the program terminates if an error occurs.

A unit may be the subject of a CLOSE statement from within any module. If the unit specified does not exist or has no file connected to it, the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

In the following example the file on UNIT 6 is closed and deleted.

```
CLOSE ( UNIT=6 , STATUS= ' DELETE ' )
```

## Data Transfer Statements

Once a unit is connected, either using a preconnection, or by executing an OPEN statement, data transfer statements may be used. The available data transfer statements include: READ, WRITE, and PRINT. The general form for these data transfer statements is shown in [Chapter 3, “Fortran Statements”](#); refer to that section for details on the READ, WRITE and PRINT statements and their valid I/O control specifiers.

## Unformatted Data Transfer

Unformatted data transfer allows data to be transferred between the current record and the items specified in an input/output list. Use OPEN to open a file for unformatted output:

```
OPEN ( 2, FILE='new.dat', FORM='UNFORMATTED' )
```

The unit specified must be an external unit.

After data is transferred, the file is positioned after the last record read or written, if there is no error condition or end-of-file condition set.

### Note

---

Unformatted data transfer cannot be carried out if the file is connected for formatted input/output.

The following example shows an unformatted input statement:

```
READ ( 2, ERR=50 ) A, B
```

- On output to a file connected for direct access, the output list must not specify more values than can fit into a record. If the values specified do not fill the record the rest of the record is undefined.
- On input, the file must be positioned so that the record read is either an unformatted record or an endfile record.
- The number of values required by the input list in the input statement must be less than or equal to the number of values in the record being read. The type of each value in the record must agree with that of the corresponding entity in the input list. However one complex value may correspond to two real list entities or vice versa. If the input list item is of type CHARACTER, its length must be the same as that of the character value
- In the event of an error condition, the position of the file is indeterminate.

## Formatted Data Transfer

During formatted data transfer, data is edited to conform to a format specification, and the edited data is transferred between the items specified in the input or output statement's iolist and the file; the current record is read or written and, possibly, so are additional records. On input, the file must be positioned so that the record read is either a formatted record or an endfile record. Formatted data transfer is prohibited if the file is connected for unformatted input/output.

For variable length record formatted input, each newline character is interpreted as a record separator. On output, the I/O system writes a newline at the end of each record. If a program writes a newline itself, the single record containing the newline will appear as two records when read or backspaced over. The maximum allowed length of a record in a variable length record formatted file is 2000 characters.

## Implied DO List Input Output List

An implied DO list takes the form

```
(iolist,do-var=var1,var2,var3)
```

where the items in iolist are either items permissible in an input/output list or another implied DO list. The value do-var is an INTEGER, REAL or DOUBLE PRECISION variable and var1, var2 and var3 are arithmetic expressions of type INTEGER, REAL or DOUBLE PRECISION. Generally, do-var, var1, var2 and var3 are of type INTEGER. Should iolist occur in an input statement, the do-var cannot be used as an item in iolist. If var3 and the preceding comma are omitted, the increment takes the value 1. The list items are specified once for each iteration of the DO loop with the DO-variable being substituted as appropriate.

In the following example OXO, C(7), C(8) and C(9) are each set to 0.0. TEMP, D(1) and D(2) are set to 10.0.

```
REAL C(6),D(6)
DATA OXO,(C(I),I=7,9),TEMP,(D(J),J=1,2)/4*0.0,3*10.0/
```

The following two statements have the same effect.

```
READ *,A,B,(R(I),I=1,4),S
READ *,A,B,R(1),R(2),R(3),R(4),S
```

## Format Specifications

Format requirements may be given either in an explicit FORMAT statement or alternatively, as fields within an input/output statement (as values in character variables, arrays or other character expressions within the input/output statement).

When a format identifier in a formatted input/output statement is a character array name or other character expression, the leftmost characters must be defined with character data that constitute a format specification when the statement is executed. A character format specification is enclosed in parentheses. Blanks may precede the left parenthesis. Character data may follow the right-hand parenthesis and has no effect on the format specification. When a character array name is used as a format identifier, the length of the format specification can exceed the length of the first element of the array; a character array format specification is considered to be an ordered concatenation of all the array elements. When a character array element is used as a format identifier the length must not exceed that of the element used.

The FORMAT statement has the form:

```
FORMAT (list-of-format-requirements)
```

The list of format requirements can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.
- Non-repeatable editor commands.
- A format specification list enclosed in parentheses, optionally preceded by an integer constant which defines the number of repeats.

Each action of format control depends on a FORMAT specified edit code and the next item in the input/output list used. If an input/output list contains at least one item, there must be at least one repeatable edit code in the format specification. An empty format specification FORMAT( ) can only be used if no list items are specified. In such a case, one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. When a repeat specification is used, the appropriate item is repeated the required number of times.

Each repeatable edit code has a corresponding item in the iolist; however when a list item is of type complex two edit codes of F, E, D or G are required. The edit codes P, X, T, TL, TR, S, SP, SS, H, BN, BZ, /, : and apostrophe act directly on the record and have no corresponding item in the input/output list.

The file is positioned after the last character read or written when the edit codes I, F, E, D, G, L, A, H or apostrophe are processed. If the specified unit is a printer then the first character of the record is used to control the vertical spacing as shown in the following table:

Table 5.2. Format Character Controls for a Printer

Character	Vertical Spacing
Blank	One line
0	Two lines
1	To first line on next page
+	No advance

### A Format Control – Character Data

The A specifier transfers characters. The A specifier has the form:

```
Aw
```

When the optional width field, w, is not specified, the width is determined by the size of the data item.

On output, if l is the length of the character item and w is the field width, then the following rules apply:

- If  $w > l$ , output with  $w-l$  blanks before the character.
- If  $w < l$ , output leftmost w characters.

On input, if l is the length of the character I/O item and w is the field width, then the following rules apply:

- If  $w > l$ , rightmost l characters from the input filed.
- If  $w < l$ , leftmost w characters from the input filed and followed by  $l - w$  blanks.

You can also use the A format specifier to process data types other than CHARACTER. For types other than CHARACTER, the number of characters supplied for input/output equals the size in bytes of the data allocated to the data type. For example, an INTEGER\*4 value is represented with 4 characters and a LOGICAL\*2 is represented with 2 characters.

The following shows a simple example that reads two CHARACTER arrays from the file data.src:

```
CHARACTER STR1*8, STR2*12
OPEN(2, FILE='data.src')
READ(2, 10) STR1, STR2
10 FORMAT ( A8, A12 )
```

## B Format Control – Binary Data

The B field descriptor transfers binary values and can be used with any integer data type. The edit descriptor has the form:

```
Bw[ .m]
```

where  $w$  specifies the field width and  $m$  indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) binary characters only (0 or 1). An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the B field descriptor transfers the binary values of the corresponding I/O list element, right-justified, to an external field that is  $w$  characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If  $m$  is present, the external field consists of at least  $m$  digits, and is zero-filled on the left if necessary.
- If  $m$  is zero, and the internal representation is zero, the external field is blank-filled.

## D Format Control – Real Double Precision Data with Exponent

The D specifier transfers real values for double precision data with a representation for an exponent. The form of the D specifier is:

```
Dw . d
```

where  $w$  is the field width and  $d$  the number of digits in the fractional part.

For input, the same conditions apply as for the F specifier described later in this chapter.

For output, the scale factor  $k$  controls the decimal normalization. The scale factor  $k$  is the current scale factor specified by the most recent P format control.

- If one hasn't been specified, the default is zero (0).
- If  $-d < k \leq 0$ , the output file contains leading zeros and  $d - |k|$  significant digits after the decimal point.
- If  $0 < k < d + 2$ , there are exactly  $|k|$  significant digits to the left of the decimal point and  $d - k + 1$  significant digits to the right of the decimal point.
- Other values of  $k$  are not allowed.

For example:

```
DOUBLE PRECISION VAL1
VAL1 = 141.8835
WRITE( *, 20) VAL1
20 FORMAT ( D10.4 )
```

produces the following:

```
0.1418D+03
```



## E Format Control – Real Single Precision Data with Exponent

The E specifier transfers real values for single precision data with an exponent. The E format specifier has two basic forms:

```
Ew . d
Ew . dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For input the same conditions apply as for F editing.

For output the scale factor controls the decimal normalization as in the D specifier.

## EN Format Control

The EN specifier transfers real values using engineering notation.

```
ENw . d
ENw . dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

On output, the number is in engineering notation where the exponent is divisible by 3 and the absolute value of the significand is  $1000 > |\text{significand}| \geq 1$ . This format is the same as the E format descriptor, except for restrictions on the size of the exponent and the significand.

## ES Format Control

The ES specifier transfers real values in scientific notation. The ES format specifier has two basic forms:

```
ESw . d
ESw . dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For output, the scale factor controls the decimal normalization as in the D specifier.

On output, the number is presented in scientific notation, where the absolute value of the significand is  $10 > |\text{significand}| \geq 1$ .

## F Format Control - Real Single Precision Data

The F specifier transfers real values. The form of the F specifier is:

```
Fw . d
```

where w is the field width and d is the number of digits in the fractional part.

On input, if the field does not contain a decimal digit or an exponent, right-hand d digits, with leading zeros, are interpreted as being the fractional part.

On output, a leading zero is only produced to the left of the decimal point if the value is less than one.

## G Format Control

The G format specifier provides generalized editing of real data. The G format has two basic forms:

```
Gw . d
Gw . dEe
```

The specifier transfers real values; it acts like the F format control on input and depending on the value's magnitude, like E or F on output. The magnitude of the data determines the output format. For details on the actual format used, based on the magnitude, refer to the ANSI FORTRAN Standard (Section 13.5.9.2.3 G Editing).

## I Format Control – Integer Data

The I format specifier transfers integer values. The I format specifier has two basic forms:

```
Iw
Iw . m
```

where *w* is the field width and *m* is the minimum field width on output, including leading zeros. If present, *m* must not exceed width *w*.

On input, the external field to be input must contain (unsigned) decimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the I format descriptor transfers the decimal values of the corresponding I/O list element, right-justified, to an external field that is *w* characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary.
- If *m* is zero, and the internal representation is zero, the external field is blank-filled.

## L Format Control – Logical Data

The L format control transfers logical data of field width *w*:

```
Lw
```

On input, the list item will become defined with a logical value; the field consists of optional blanks, followed by an optional decimal point followed by T or F. The values `.TRUE.` or `.FALSE.` may also appear in the input field.

The output field consists of *w*-1 blanks followed by T or F as appropriate.

## Quote Format Control

Quote editing prints a character constant. The format specifier writes the characters enclosed between the quotes and cannot be used on input. The field width is that of the characters contained within quotes (you can also use apostrophes to enclose the character constant).

To write an apostrophe (or quote), use two consecutive apostrophes (or quotes).

For example:

```
WRITE ( *, 101)
101 FORMAT ( 'Print an apostrophe ' and end.')
```

Produces:

```
Print an apostrophe ' and end.
```

Similarly, you can use quotes, for example:

```
WRITE ( *, 102)
102 FORMAT ( "Print a line with a " and end.")
```

Produces:

```
Print a line with a " and end.
```

### BN Format Control – Blank Control

The BN and BZ formats control blank spacing. BN causes all embedded blanks except leading blanks in numeric input to be ignored, which has the effect of right-justifying the remainder of the field. Note that a field of all blanks has the value zero. Only input statements and I, F, E, D and G editing are affected.

BZ causes all blanks except leading blanks in numeric input to be replaced by zeros. Only input statements and I, F, E, D and G editing are affected.

### H Format Control – Hollerith Control

The H format control writes the n characters following the H in the format specification and cannot be used on input.

The basic form of this format specification is:

```
nHc1cn...
```

where n is the number of characters to print and c1 through cn are the characters to print.

### O Format Control Octal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with an integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.

- If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.
- If m is zero, and the internal representation is zero, the external field is blank-filled.

### P Format Specifier – Scale Control

The P format specifier is the scale factor format.

```
kP
```

This specifier is applied as follows.

- With F, E, D and G editing on input and F editing on output, the external number equals the internal number multiplied by  $10^{**k}$ .
- If there is an exponent in the field on input, editing with F, E, D and G the scale factor has no effect.
- On output with E and D editing, the basic real constant part of the number is multiplied by  $10^{**k}$  and the exponent reduced by k.
- On output with G editing, the effect of the scale factor is suspended unless the size of the datum to be edited is outside the range permitted for F editing.
- On output if E editing is required, the scale factor has the same effect as with E output editing.

The following example uses a scale factor.

```
DIMENSION
A(6)
DO 10 I = 1,6
10 A(I) = 25.
TYPE 100,A
100 FORMAT(' ',F8.2,2PF8.2,F8.2)
```

This example produces this output:

```
25.00
2500.00 2500.00 2500.00 2500.00 2500.00
```

#### Note

---

The effect of the scale factor continues until another scale factor is used.

### Q Format Control - Quantity

The Q edit descriptor calculates the number of characters remaining in the input record and stores that value in the next I/O list item. On output, the Q descriptor skips the next I/O item.

### S Format Control – Sign Control

The S format specifier restores the default processing for writing a plus; the default is SS processing.

SP forces the processor to write a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

SS stops the processor from writing a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

## T, TL and X Format Controls – Spaces and Tab Controls

The T specifier controls which portion of a record in an iolist value is read from or written to a file. The general form, which specifies that the nth value is to be written to or from a record, is as follows:

```
Tn
```

The TL form specifies the relative position to the left of the data to be read or written, and specifies that the nth character to the left of the current position is to be written to or from the record. If the current position is less than or equal to n, the transmission will begin at position one of the record.

```
TLn
```

The TR form specifies the relative position to the right of the data to be read or written, and specifies that the nth character to the right of the current position is to be written to or from the record.

```
TRn
```

The X control specifies a number of characters to skip forward, and that the next character to be written to or from is n characters forward from the current position.

```
nX
```

The following example uses the X format specifier:

```
NPAGE = 19
WRITE ( 6, 90) NPAGE
90 FORMAT('1PAGE NUMBER ,I2, 16X, 'SALES REPORT, Cont.')
```

produces:

```
PAGE NUMBER 19 SALES REPORT, Cont.
```

The following example shows use of the T format specifier:

```
PRINT 25
25 FORMAT (T41, 'COLUMN 2', T21, 'COLUMN 1')
```

produces:

```
COLUMN 1 COLUMN 2
```

## Z Format Control Hexadecimal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with any integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.

- If the value is too large for the field, the entire field is filled with asterisks.
- If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary.
- If *m* is zero, and the internal representation is zero, the external field is blank-filled.

### Slash Format Control / – End of Record

The slash (/) control indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the rest of the current record is skipped and the file positioned at the start of the next record.

On output a new record is created which becomes the last and current record.

- For an internal file connected for direct access, the record is filled with blank characters.
- For a direct access file, the record number is increased by one and the file is positioned at the start of the record.

#### Note

---

Multiple slashes are permitted, thus multiple records are skipped.

### The : Format Specifier – Format Termination

The (:) control terminates format control if there are no more items in the input/output list. It has no effect if there are any items left in the list.

### \$ Format Control

The \$ field descriptor allows the programmer to control carriage control conventions on output. It is ignored on input. For example, on terminal output, it can be used for prompting.

The form of the \$ field descriptor is:

```
$
```

### Variable Format Expressions

Variable format expressions, <expr>, are supported. They provide a means for substituting run-time expressions for the field width, other parameters for the field and edit descriptors in a FORMAT statement (except for the H field descriptor and repeat counts).

Variable format expressions are enclosed in "<" and ">" and are evaluated each time they are encountered in the scan of a format. If the value of a 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.

## Non-advancing Input and Output

Non-advancing input/output is character-oriented and applies to sequential access formatted external files. The file position is after the last character read or written and not automatically advanced to the next record.

For non-advancing input/output, use the `ADVANCE='NO'` specifier. Two other specifiers apply to non-advancing IO: `EOR` applies when end of record is detected and `SIZE` returns the number of characters read.

## List-directed formatting

List-directed formatting is an abbreviated form of input/output that does not require the use of a format specification. The type of the data determines how a value is read/written. On output, it is not always accurate enough for certain ranges of values. The characters in a list-directed record constitute a sequence of values which cannot contain embedded blanks except those permitted within a character string.

To use list-directed input/output formatting, specify a `*` for the list of format requirements, as illustrated in the following example that uses list-directed output:

```
READ( 1, * ) VAL1, VAL2
```

### List-directed input

The form of the value being input must be acceptable for the type of item in the iolist. Blanks must not be used as zeros nor be embedded in constants except in a character constant or within a type complex form contained in parentheses.

Table 5.3. List Directed Input Values

Input List Type	Form
Integer	A numeric input field.
Real	A numeric input field suitable for F editing with no fractional part unless a decimal point is used.
Double precision	Same as for real.
Complex	An ordered pair of numbers contained within parentheses as shown: (real part, imaginary part).
Logical	A logical field without any slashes or commas.
Character	A non-empty character string within apostrophes. A character constant can be continued on as many records as required. Blanks, slashes and commas can be used.

A null value has no effect on the definition status of the corresponding iolist item. A null value cannot represent just one part of a complex constant but may represent the entire complex constant. A slash encountered as a value separator stops the execution of that input statement after the assignment of the previous value. If there are further items in the list, they are treated as if they are null values.

Commas may be used to separate the input values. If there are consecutive commas, or if the first non-blank character of a record is a comma, the input value is a null value. Input values may also be repeated.

In the following example of list-directed formatting, assume that `A` and `K` are defined as follows and all other variables are undefined.

```
A= -1.5
K= 125
```

Suppose that you have an input file the contains the following record, where the / terminates the input and consecutive commas indicate a null:

```
10,-14,25.2,-76,313,,29/
```

Further suppose that you use the following statement to read in the list from the input file:

```
READ * I, J, X, Y, Z, A, C, K
```

The variables are assigned the following values by the list-directed input/output mechanism:

```
I=10          J=-14          X=25.2        Y=-76.0
Z=313.0       A=-1.5         C=29          K=125
```

In the example the value for A does not change because the input record is null. Input is terminated with the / so no input is read for K, so the program assumes null and K retains its previous value.

## List-directed output

List directed input/output is an abbreviated form of formatted input/output that does not require the use of a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to or from the file, using a default, and not necessarily accurate format specification. The data type of each item appearing in the iolist is formatted according to the rules in the following table:

Table 5.4. Default List Directed Output Formatting

Data Type	Default Formatting
BYTE	I5
INTEGER*2	I7
INTEGER*4	I12
INTEGER*8	I24
LOGICAL*1	I5 (L2 <sup>a</sup> )
LOGICAL*2	L2
LOGICAL*4	L2
LOGICAL*8	L2
REAL*4	G15.7e2
REAL*8	G25.16e3
COMPLEX*8	(G15.7e2, G15.7e2)
COMPLEX*16	(G25.16e3, G25.16e3)
CHAR *n	An

<sup>a</sup>This format is applied when the option `-Munixlogical` is selected when compiling.

The length of a record is less than 80 characters; if the output of an item would cause the length to exceed 80 characters, a new record is created.

The following rules and guidelines apply when using list-directed output:



- New records may begin as necessary.
- Logical output constants are T for true and F for false.
- Complex constants are contained within parentheses with the real and imaginary parts separated by a comma.
- Character constants are not delimited by apostrophes and have each internal apostrophe (if any are present) represented externally by one apostrophe.
- Each output record begins with a blank character to provide carriage control when the record is printed.
- A typeless value output with list-directed I/O is output in hexadecimal form by default. There is no other octal or hexadecimal capability with list-directed I/O.

## Commas in External Field

Use of the comma in an external field eliminates the need to "count spaces" to have data match format edit descriptors. The use of a comma to terminate an input field and thus avoid padding the field is fully supported.

## Namelist Groups

The NAMELIST statement allows for the definition of namelist groups. A namelist group allows for a special type of formatted input/output, where data is transferred between a named group of data items defined in a NAMELIST statement and one or more records in a file.

The general form of a namelist statement is:

```
NAMELIST /group-name/ namelist [[,] /group-name/ namelist ]...
```

where:

`group-name`

is the name of the namelist group.

`namelist`

is the list of variables in the namelist group.

## Namelist Input

Namelist input is accomplished using a READ statement by specifying a namelist group as the input item. The following statement shows the format:

```
READ ([unit=] u, [NML=] namelist-group [,control-information])
```

One or more records are processed which define the input for items in the namelist group.

The records are logically viewed as follows:

```
$group-name item=value [,item=value]... $ [END]
```

The following rules describe these input records:

- The start or end delimiter (\$) may be an ampersand (&).

- The start delimiter must begin in column 2 of a record.
- The group-name begins immediately after the start delimiter.
- The spaces or tabs may not appear within the group-name, within any item, or within any constants.
- The value may be constants as are allowed for list directed input, or they may be a list of constants separated by commas (.). A list of items is used to assign consecutive values to consecutive elements of an array.
- Spaces or tabs may precede the item, the = and the constants.
- Array items may be subscripted.
- Character items may have substrings.

## Namelist Output

Namelist output is accomplished using a READ statement by specifying a namelist group as the output item. The following statement shows the format:

```
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

The records output are logically viewed as follows:

```
$group-name  
item = value  
$ [END]
```

The following rules describe these output records:

- One record is output per value.
- Multiple values are separated by a comma (.).
- Values are formatted according to the rules of the list-directed write. Exception: character items are delimited by an apostrophe (').
- An apostrophe (') or a quote (") in the value is represented by two consecutive apostrophes or quotes.

# Chapter 6. Fortran Intrinsic

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

This chapter lists the FORTRAN 77 and Fortran 90/95 intrinsics and subroutines, intrinsics defined in the HPF Language Specification, and CM Fortran intrinsics and Fortran 2003 intrinsic modules. The Fortran processor, rather than the user or a third party, provides the intrinsic functions and intrinsic modules.

## Intrinsics Support

The tables in this section contain the FORTRAN 77, Fortran 90/95 and Fortran 2003 intrinsics that are supported.

Table 6.1. Fortran 90/95 Bit Manipulation Functions and Subroutines

Generic Name	Purpose	Num. Args	Argument Type	Result Type
AND	Performs a logical AND on corresponding bits of the arguments.	2	ANY type except CHAR or COMPLEX	
BIT_SIZE	Return the number of bits (the precision) of the integer argument.	1	INTEGER	INTEGER
BTEST	Tests the binary value of a bit in a specified position of an integer argument.	2	INTEGER, INTEGER	LOGICAL
IAND	Perform a bit-by-bit logical AND on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
IBCLR	Clears one bit to zero.	2	INTEGER, INTEGER $\geq 0$	INTEGER
IBITS	Extracts a sequence of bits.		INTEGER, INTEGER $\geq 0$ , INTEGER $\geq 0$	INTEGER

<b>Generic Name</b>	<b>Purpose</b>	<b>Num. Args</b>	<b>Argument Type</b>	<b>Result Type</b>
IBSET	Sets one bit to one.	2	INTEGER, INTEGER $\geq 0$	INTEGER
IEOR	Perform a bit-by-bit logical exclusive OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
IOR	Perform a bit-by-bit logical OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
ISHFT	Perform a logical shift.	2	INTEGER, INTEGER	INTEGER
ISHFTC	Perform a circular shift of the rightmost bits.	2 or 3	INTEGER, INTEGER INTEGER, INTEGER, INTEGER	INTEGER
LSHIFT	Perform a logical shift to the left.	2	INTEGER, INTEGER	INTEGER
MVBITS	Copies bit sequence	5	INTEGER(IN), INTEGER(IN), INTEGER(IN), INTEGER(INOUT), INTEGER(IN)	none
NOT	Perform a bit-by-bit logical complement on the argument.	2	INTEGER	INTEGER
OR	Performs a logical OR on each bit of the arguments.	2	ANY type except CHAR or COMPLEX	
POPCNT	Return the number of one bits.	1	INTEGER or bits	INTEGER
POPPAR	Return the bitwise parity.	1	INTEGER or bits	INTEGER
RSHIFT	Perform a logical shift to the right.	2	INTEGER, INTEGER	INTEGER
SHIFT	Perform a logical shift.	2	Any type except CHAR or COMPLEX, INTEGER	
XOR	Performs a logical exclusive OR on each bit of the arguments.	2	INTEGER, INTEGER	INTEGER
ZEXT	Zero-extend the argument.	1	INTEGER or LOGICAL	INTEGER

The functions in the following table are specific to Fortran 90/95 unless otherwise specified.

Table 6.2. Elemental Character and Logical Functions

<b>Generic Name</b>	<b>Purpose</b>	<b>Num. Args</b>	<b>Argument Type</b>	<b>Result Type</b>
ACHAR	Return character in specified ASCII collating position.	1	INTEGER	CHARACTER
ADJUSTL	Left adjust string	1	CHARACTER	CHARACTER

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ADJUSTR	Right adjust string	1	CHARACTER	CHARACTER
CHAR (f77)	Return character with specified ASCII value.	1	LOGICAL*1 INTEGER	CHARACTER CHARACTER
IACHAR	Return position of character in ASCII collating sequence.	1	CHARACTER	INTEGER
ICHAR	Return position of character in the character set's collating sequence.	1	CHARACTER	INTEGER
INDEX	Return starting position of substring within first string.	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
LEN	Returns the length of string	1	CHARACTER	INTEGER
LEN_TRIM	Returns the length of the supplied string minus the number of trailing blanks.	1	CHARACTER	INTEGER
LGE	Test the supplied strings to determine if the first string is lexically greater than or equal to the second.	2	CHARACTER, CHARACTER	LOGICAL
LGT	Test the supplied strings to determine if the first string is lexically greater than the second.	2	CHARACTER, CHARACTER	LOGICAL
LLE	Test the supplied strings to determine if the first string is lexically less than or equal to the second.	2	CHARACTER, CHARACTER	LOGICAL
LLT	Test the supplied strings to determine if the first string is lexically less than the second.	2	CHARACTER, CHARACTER	LOGICAL
LOGICAL	Logical conversion	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
SCAN	Scan string for characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
VERIFY	Determine if string contains all characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER

Table 6.3. Fortran 90/95 Vector/Matrix Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
DOT_PRODUCT	Perform dot product on two vectors	2	NONCHAR*K, NONCHAR*K	NONCHAR*K
MATMUL	Perform matrix multiply on two matrices	2	NONCHAR*K, NONCHAR*K	NONCHAR*K

Table 6.4. Fortran 90/95 Array Reduction Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ALL	Determine if all array values are true	1	LOGICAL	LOGICAL
		2	LOGICAL, INTEGER	LOGICAL
ANY	Determine if any array value is true	1	LOGICAL	LOGICAL
		2	LOGICAL, INTEGER	LOGICAL
COUNT	Count true values in array	1	LOGICAL	INTEGER
		2	LOGICAL, INTEGER	INTEGER
MAXLOC	Determine position of array element with maximum value	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	
		2	INTEGER, INTEGER	
		3	INTEGER, INTEGER, LOGICAL	
		1	REAL	
		2	REAL, LOGICAL	
		2	REAL, INTEGER	
		3	REAL, INTEGER, LOGICAL	
MAXVAL	Determine maximum value of array elements	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
		3	REAL, INTEGER, LOGICAL	REAL
MINLOC	Determine position of array element with minimum value	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	
		2	INTEGER, INTEGER	
		3	INTEGER, INTEGER, LOGICAL	
		1	REAL	
		2	REAL, LOGICAL	
		2	REAL, INTEGER	
		3	REAL, INTEGER, LOGICAL	

Generic Name	Purpose	Num. Args	Argument Type	Result Type
MINVAL	Determine minimum value of array elements	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
PRODUCT	Calculate the product of the elements of an array	1	NUMERIC	NUMERIC
		2	NUMERIC, LOGICAL	
		2	NUMERIC, INTEGER	
		3	NUMERIC, INTEGER, LOGICAL	
SUM	Calculate the sum of the elements of an array	1	NUMERIC	NUMERIC
		2	NUMERIC, LOGICAL	
		2	NUMERIC, INTEGER	
		3	NUMERIC, INTEGER, LOGICAL	

Table 6.5. Fortran 90/95 String Construction Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
REPEAT	Concatenate copies of a string	2	CHARACTER, INTEGER	CHARACTER
TRIM	Remove trailing blanks from a string	1	CHARACTER	CHARACTER

Table 6.6. Fortran 90/95 Array Construction/Manipulation Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
CSHIFT	Perform circular shift on array	2	ARRAY, INTEGER	ARRAY <sup>a</sup>
		3	ARRAY, INTEGER, INTEGER	ARRAY <sup>a</sup>
EOSHIFT	Perform end-off shift on array	2	ARRAY, INTEGER	ARRAY <sup>a</sup>
		3	ARRAY, INTEGER, any <sup>a</sup>	ARRAY <sup>a</sup>
		3	ARRAY, INTEGER, INTEGER	ARRAY <sup>a</sup>
		4	ARRAY, INTEGER, any <sup>a</sup> , INTEGER	ARRAY <sup>a</sup>
MERGE	Merge two arguments based on logical mask	3	any, any <sup>a</sup> , LOGICAL	any <sup>a</sup>
PACK	Pack array into rank-one array	2	ARRAY, LOGICAL	ARRAY <sup>a</sup>
		3	ARRAY, LOGICAL, VECTOR <sup>a</sup>	ARRAY <sup>a</sup>

Generic Name	Purpose	Num. Args	Argument Type	Result Type
RESHAPE	Change the shape of an array	2 3 3 4	ARRAY, INTEGER ARRAY, INTEGER, ARRAY <sup>a</sup> ARRAY, INTEGER, INTEGER ARRAY, INTEGER, ARRAY <sup>a</sup> , INTEGER	ARRAY <sup>a</sup>
SPREAD	Replicates an array by adding a dimension	3	any, INTEGER, INTEGER	ARRAY <sup>a</sup>
TRANSPOSE	Transpose an array of rank two	1	ARRAY	ARRAY <sup>a</sup>
UNPACK	Unpack a rank-one array into an array of multiple dimensions	3	VECTOR, LOGICAL, ARRAY <sup>a</sup>	ARRAY <sup>a</sup>

<sup>a</sup>Must be of the same type as the first argument.

Table 6.7. Fortran 90/95 General Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ASSOCIATED	Determine association status	12	POINTER, POINTER,..., POINTER, TARGET	LOGICAL LOGICAL
KIND	Determine argument's kind	1	any intrinsic type	INTEGER
PRESENT	Determine presence of optional argument	1	any	LOGICAL

Table 6.8. Fortran 90/95 Numeric Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
DIGITS	Determine number of significant digits	1 1	INTEGER REAL	INTEGER
EPSILON	Smallest representable number	1	REAL	REAL
HUGE	Largest representable number	1 1	INTEGER REAL	INTEGER REAL
MAXEXPONENT	Value of maximum exponent	1	REAL	INTEGER



Generic Name	Purpose	Number of Args	Argument Type	Result Type
MINEXPONENT	Value of minimum exponent	1	REAL	INTEGER
PRECISION	Decimal precision	1 1	REAL COMPLEX	INTEGER INTEGER
RADIX	Base of model	1 1	INTEGER REAL	INTEGER INTEGER
RANGE	Decimal exponent range	1 1 1	INTEGER REAL COMPLEX	INTEGER INTEGER INTEGER
SELECTED_INT_KIND	Kind type titlemeter in range	1	INTEGER	INTEGER
SELECTED_REAL_KIND	Kind type titlemeter in range	1 2	INTEGER INTEGER, INTEGER	INTEGER INTEGER
TINY	Smallest representable positive number	1	REAL	REAL

Table 6.9. Fortran 90/95 Array Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ALLOCATED	Determine if array is allocated	1	ARRAY	LOGICAL
LBOUND	Determine lower bounds	1 2	ARRAY ARRAY, INTEGER	INTEGER
SHAPE	Determine shape	1	any	INTEGER
SIZE	Determine number of elements	1 2	ARRAY ARRAY, INTEGER	INTEGER
UBOUND	Determine upper bounds	1 2	ARRAY ARRAY, INTEGER	INTEGER

Table 6.10. Fortran 90/95 Subroutines

Generic Name	Purpose	Number of Args	Argument Type
CPU_TIME	Returns processor time	1	REAL (OUT)

Generic Name	Purpose	Number of Args	Argument Type
DATE_AND_TIME	Returns date and time	4 (optional)	DATE (CHARACTER, OUT) TIME (CHARACTER, OUT) ZONE (CHARACTER, OUT) VALUES (INTEGER, OUT)
RANDOM_NUMBER	Generate pseudo-random numbers	1	REAL (OUT)
RANDOM_SEED	Set or query pseudo-random number generator	0 1 1 1	SIZE (INTEGER, OUT) PUT (INTEGER ARRAY, IN) GET (INTEGER ARRAY, OUT)
SYSTEM_CLOCK	Query real time clock	3 (optional)	COUNT (INTEGER, OUT) COUNT_RATE (REAL, OUT) COUNT_MAX (INTEGER, OUT)

Table 6.11. Fortran 90/95 Transfer Function

Generic Name	Purpose	Number of Args	Argument Type	Result Type
TRANSFER	Change type but maintain bit representation	23 3	any, any any, any, INTEGER	any <sup>a</sup>

<sup>a</sup>Must be of the same type as the second argument.

Table 6.12. Arithmetic Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ABS	Return absolute value of the supplied argument.	1	INTEGER REAL COMPLEX	INTEGER REAL COMPLEX
ACOS	Return the arccosine (in radians) of the specified value	1	REAL	REAL
ACOSD	Return the arccosine (in degrees) of the specified value	1	REAL	REAL
AIMAG	Return the value of the imaginary part of a complex number.	1	COMPLEX	REAL

<b>Generic Name</b>	<b>Purpose</b>	<b>Num. Args</b>	<b>Argument Type</b>	<b>Result Type</b>
AINT	Truncate the supplied value to a whole number.	2	REAL, INTEGER	REAL
AND	Performs a logical AND on corresponding bits of the arguments.	2	ANY type except CHAR or COMPLEX	
ANINT	Return the nearest whole number to the supplied argument.	2	REAL, INTEGER	REAL
ASIN	Return the arcsine (in radians) of the specified value	1	REAL	REAL
ASIND	Return the arcsine (in degrees) of the specified value	1	REAL	REAL
ATAN	Return the arctangent (in radians) of the specified value	1	REAL	REAL
ATAN2	Return the arctangent (in radians) of the specified pair of values.	2	REAL, REAL	REAL
ATAN2D	Return the arctangent (in degrees) of the specified pair of values	1	REAL, REAL	REAL
ATAND	Return the arctangent (in degrees) of the specified value	1	REAL	REAL
CEILING	Return the least integer greater than or equal to the supplied real argument.	2	REAL, KIND	INTEGER
CMPLX	Convert the supplied argument or arguments to complex type.	2 3	INTEGER, REAL, or COMPLEX; INTEGER, REAL, or COMPLEX; INTEGER, REAL, or COMPLEX; INTEGER or REAL KIND	COMPLEX
COMPL	Performs a logical complement on the argument.	1	ANY, except CHAR or COMPLEX	
COS	Return the cosine (in radians) of the specified value	1	REAL COMPLEX	REAL
COSD	Return the cosine (in degrees) of the specified value	1	REAL COMPLEX	REAL

<b>Generic Name</b>	<b>Purpose</b>	<b>Num. Args</b>	<b>Argument Type</b>	<b>Result Type</b>
COSH	Return the hyperbolic cosine of the specified value	1	REAL	REAL
DBLE	Convert to double precision real.		INTEGER, REAL, or COMPLEX	REAL
DCMPLX	Convert the supplied argument or arguments to double complex type.	1 2	INTEGER, REAL, or COMPLEX INTEGER, REAL	DOUBLE COMPLEX
DPROD	Double precision real product.	2	REAL, REAL	REAL (double precision)
EQV	Performs a logical exclusive NOR on the arguments.	2	ANY except CHAR or COMPLEX	
EXP	Exponential function.	1	REAL COMPLEX	REAL COMPLEX
EXPONENT	Return the exponent part of a real number.	1	REAL	INTEGER
FLOOR	Return the greatest integer less than or equal to the supplied real argument.	1 2	REAL REAL, KIND	REAL KIND
FRACTION	Return the fractional part of a real number.	1	REAL	INTEGER
IINT	Converts a value to a short integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
ININT	Returns the nearest short integer to the real argument.	1	REAL	INTEGER
INT	Converts a value to integer type.	1 2	INTEGER, REAL, or COMPLEX INTEGER, REAL, or COMPLEX; KIND	INTEGER
INT8	Converts a real value to a long integer type.	1	REAL	INTEGER
IZEXT	Zero-extend the argument.	1	LOGICAL or INTEGER	INTEGER
JINT	Converts a value to an integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
JNINT	Returns the nearest integer to the real argument.	1	REAL	INTEGER
KNINT	Returns the nearest integer to the real argument.	1	REAL	INTEGER (long)

<b>Generic Name</b>	<b>Purpose</b>	<b>Num. Args</b>	<b>Argument Type</b>	<b>Result Type</b>
LOG	Returns the natural logarithm.	1	REAL or COMPLEX	REAL
LOG10	Returns the common logarithm.	1	REAL	REAL
MAX	Return the maximum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as Argument Type
MIN	Return the minimum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as Argument Type
MOD	Find the remainder.	2 or more	INTEGER or REAL, INTEGER or REAL (all of same kind)	Same as Argument Type
MODULO	Return the modulo value of the arguments.	2 or more	INTEGER or REAL, INTEGER or REAL (all of same kind)	Same as Argument Type
NEAREST	Returns the nearest different machine representable number in a given direction.	2	REAL, non-zero REAL	REAL
NEQV	Performs a logical exclusive OR on the arguments.	2	ANY except CHAR or COMPLEX	
NINT	Converts a value to integer type.	1 2	REAL REAL, KIND	INTEGER
REAL	Convert the argument to real.	1 2	IINTEGER, REAL, or COMPLEX INTEGER, REAL, or COMPLEX; KIND	REAL
RRSPACING	Return the reciprocal of the relative spacing of model numbers near the argument value.	1	REAL	REAL
SET_EXPONENT	Returns the model number whose fractional part is the fractional part of the model representation of the first argument and whose exponent part is the second argument.	2	REAL, INTEGER	REAL
SIGN	Return the absolute value of A times the sign of B.	2	INTEGER or REAL, INTEGER or REAL (of same kind)	Same as Argument
SIN	Return the sine (in radians) of the specified value	1	REAL COMPLEX	REAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
SIND	Return the sine (in degrees) of the specified value	1	REAL COMPLEX	REAL
SINH	Return the hyperbolic sine of the specified value	1	REAL	REAL
SPACING	Return the relative spacing of model numbers near the argument value.	1	REAL	REAL
SQRT	Return the square root of the argument.	1	REAL COMPLEX	REAL COMPLEX
TAN	Return the tangent (in radians) of the specified value	1	REAL	REAL
TAND	Return the tangent (in degrees) of the specified value	1	REAL	REAL
TANH	Return the hyperbolic tangent of the specified value	1	REAL	REAL

Table 6.13. Fortran 2003 Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
EXTENDS_TYPE_OF	Determines whether the dynamic type of A is an extension type of the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR
GET_COMMAND_ARGUMENT	Returns the specified command line argument of the command that invoked the program.	1 to 4	INTEGER plus optionally: CHAR, INTEGER, INTEGER	A command argument
GET_COMMAND	Returns the entire command line that was used to invoke the program.	0 to 3	CHAR, INTEGER, INTEGER	A command line
GET_ENVIRONMENT_VARIABLE	Returns the value of the specified environment variable.	1 to 5	CHAR, CHAR, INTEGER, INTEGER, LOGICAL	
IS_IOSTAT_END	Test whether a variable has the value of the I/O status: "end of file".	1	INTEGER	LOGICAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
IS_IOSTAT_EOR	Test whether a variable has the value of the I/O status: “end of record”.	1	INTEGER	LOGICAL
LEADZ	Counts the number of leading zero bits.	1	INTEGER or bits	INTEGER
MOVE_ALLOC	Moves an allocation from one allocatable object to another.	2	Any - of same type and rank	none
NEW_LINE	Return the newline character.	1	CHARACTER	CHARACTER
SAME_TYPE_AS	Determines whether the dynamic type of A is the same as the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR
SCALE	Return the value $X * b^i$ where b is the base of the number system in use for X.	2	REAL, INTEGER	REAL

Table 6.14. Miscellaneous Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
LOC	Return address of argument	1	NUMERIC	INTEGER
NULL	Assign disassociated status	0		POINTER
		1	POINTER	POINTER

## ABS

Determine the absolute value of the supplied argument.

F77

### Synopsis

```
ABS ( A )
```

### Argument

The argument A must be of type integer, real, or complex.

## Return Value

The return type for integer is integer, for real is real, and for complex is real.

## ACHAR

Return the character in the ASCII collating position specified by the argument.

## F90

## Synopsis

```
ACHAR(I [, KIND])
```

## Argument

The argument I must be of type integer. The optional KIND argument is a scalar integer initialization expression.

## Return Value

A single character. If the KIND argument is present, the return value is of the specified kind; otherwise, the return value is of type default character.

## ACOS

Return the arccosine of the specified value.

## F77

## Synopsis

```
ACOS(X)
```

## Arguments

The argument X must be a real value.

## Return Value

The real value representing the arccosine in radians.

## ACOSD

Return the arccosine (in degrees) of the specified value.

## F77

## Synopsis

```
ACOSD(X)
```



## Arguments

The argument X must be a real value.

## Return Value

The real value representing the arccosine in degrees.

## ADJUSTL

Adjust string to the left, removing all leading blanks and inserting trailing blanks.

## F90

## Synopsis

```
ADJUSTL ( STR )
```

## Arguments

The argument STR is the string to be adjusted.

## Return Value

String of same length and kind as the argument with leading blanks removed and the same number of trailing blanks added.

## ADJUSTR

Adjust string to the right, removing all trailing blanks and inserting leading blanks.

## F90

## Synopsis

```
ADJUSTR ( STR )
```

## Arguments

The argument STR is the string to be adjusted.

## Return Value

String of same length and kind as the argument with trailing blanks removed and the same number of leading blanks added.

## AIMAG

Determine the value of the imaginary part of a complex number.

AINT

F77

## Synopsis

```
AIMAG(Z)
```

## Arguments

The argument *Z* must be complex.

## Return Value

A real value representing the imaginary part of the supplied argument.

## AINT

Truncate the supplied value to a whole number.

F77

## Synopsis

```
AINT(A [, KIND])
```

## Arguments

The argument *A* is of type real. The optional *KIND* argument is a scalar integer initialization expression.

## Return Value

A real value that is equal to the largest integer that is not greater than the supplied argument. If the *KIND* argument is present, the result is of that kind.

## ALL

Determine if all the values in the supplied argument are logical true.

F90

## Synopsis

```
ALL(MASK [, DIM])
```

## Arguments

The argument *MASK* is an array of type LOGICAL. The optional argument *DIM* specifies the dimension of the array *MASK* to check.

## Return Value

If no *DIM* argument is present, the return value is a logical scalar that is true only if all values of *MASK* are true.

If the *DIM* argument is present and if *MASK* has rank one, then the return value is the same as `ALL(MASK)`.

If the DIM argument is present and MASK has rank greater than one, then the return value is an array that has rank n-1, where n is the rank of MASK. The return value is defined recursively as the value of ALL for each extent of the dimension DIM (refer to the Fortran 95 Handbook for a more detailed explanation).

## ALLOCATED

Determine if the supplied allocatable array is currently allocated.

F90

### Synopsis

```
ALLOCATED ( ARRAY )
```

### Argument

The argument ARRAY is an allocatable array.

### Return Value

Returns a logical scalar indicating whether the array is allocated.

## AND

Performs a logical AND on corresponding bits of the arguments.

F77 extension

### Synopsis

```
AND ( M, N )
```

### Arguments

The arguments M and N may be of any type except for character and complex.

### Return Value

The return value is typeless.

## ANINT

Return the nearest whole number to the supplied argument.

F77

### Synopsis

```
ANINT ( A [ , KIND ] )
```

### Arguments

The argument A is a real number. The optional argument KIND is a kind parameter.

ANY

## Return Value

The result is a real. The value is  $\text{AINT}(A+0.5)$  if  $A$  is  $> 0$  and  $\text{AINT}(A-0.5)$  if  $A$  is  $< 0$ .

If  $\text{KIND}$  is present, the result is of type  $\text{KIND}$ .

## ANY

Determine if any value in the supplied argument  $\text{MASK}$  is true.

F90

## Synopsis

```
ANY ( MASK [ , DIM ] )
```

## Arguments

The argument  $\text{MASK}$  is an array of type  $\text{LOGICAL}$ . The optional argument  $\text{DIM}$  specifies the dimension of the array  $\text{MASK}$  to check.

## Return Value

If no  $\text{DIM}$  argument is present, the return value is a logical scalar that is true if any element of  $\text{MASK}$  is true.

If the  $\text{DIM}$  argument is present and if  $\text{MASK}$  has rank one, then the return value is the same as  $\text{ANY}(\text{MASK})$ .

If the  $\text{DIM}$  argument is present and  $\text{MASK}$  has rank greater than one, then the return value is an array that has rank  $n-1$ , where  $n$  is the rank of  $\text{MASK}$ . The return value is defined recursively as the value of  $\text{ANY}$  for each extent of the dimension  $\text{DIM}$  (refer to The Fortran 95 Handbook for a more detailed explanation)

## ASIN

Return the arcsine of the specified value.

F77

## Synopsis

```
ASIN ( X )
```

## Argument

The argument  $X$  must be of type real and have absolute value  $\leq 1$ .

## Return Value

The real value representing the arcsine in radians.

## ASIND

Return the arcsine (in degrees) of the specified value.

F77

## Synopsis

`ASIND ( X )`

## Argument

The argument X must be of type real and have absolute value  $\leq 1$ .

## Return Value

The real value representing the arcsine in degrees.

**ASSOCIATED**

Determines the association status of the supplied argument or determines if the supplied pointer is associated with the supplied target.

F90

## Synopsis

`ASSOCIATED ( POINTER [ , TARGET ] )`

## Arguments

The POINTER argument is a pointer of any type. The optional argument TARGET is a pointer or a target. If it is a pointer it must not be undefined.

## Return Value

If TARGET is not supplied the function returns logical true if POINTER is associated with a target and false otherwise.

If TARGET is present and is a target, then the function returns true if POINTER is associated with TARGET and false otherwise.

If TARGET is present and is a pointer, then the function returns true if POINTER and TARGET are associated with the same target and false otherwise.

**ATAN**

Return the arctangent of the specified value.

F77

## Synopsis

`ATAN ( X )`

## Argument

The argument X must be of type real.

## Return Value

The real value representing the arctangent in radians.

## ATAN2

Return the arctangent of the specified value.

F77

## Synopsis

```
ATAN2 ( Y , X )
```

## Arguments

The arguments X and Y must be of type real.

## Return Value

A real number that is the arctangent for pairs of reals, X and Y, expressed in radians. The result is the principal value of the nonzero complex number (X,Y).

## ATAN2D

Return the arctangent (in degrees) of the specified value.

F77

## Synopsis

```
ATAN2D ( Y , X )
```

## Arguments

The arguments X and Y must be of type real.

## Return Value

A real number that is the arctangent for pairs of reals, X and Y, expressed in degrees. The result is the principal value of the nonzero complex number (X,Y).

## ATAND

Return the arctangent (in degrees) of the specified value.

F77

## Synopsis

```
ATAND ( X )
```

## Argument

The argument X must be of type real.

## Return Value

The real value representing the arctangent in degrees.

## BIT\_SIZE

Return the number of bits (the precision) of the integer argument. This function uses the standard Fortran 90/95 bit model defined in The Fortran 95 Handbook.

## F90

## Synopsis

```
BIT_SIZE(I)
```

## Argument

The argument I must be of type integer.

## Return Value

Returns an integer.

## BTEST

Tests the binary value of a bit in a specified position of an integer argument. This function uses the standard Fortran 90/95 bit model defined in The Fortran 95 Handbook.

## F77

## Synopsis

```
BTEST(I, POS)
```

## Arguments

The argument I must be of type integer. The argument POS must be an integer with a value less than or equal to the value BIT\_SIZE(I).

## Return Value

Returns a logical value representing whether the bit in position POS is true (0) or false (1).

## CEILING

Return the least integer greater than or equal to the supplied real argument.

CHAR

F90

## Synopsis

```
CEILING(A [,KIND])
```

## Argument

The argument A is a real value. The optional argument KIND is a kind parameter and was added to CEILING in Fortran 95.

## Return Value

An integer.

If KIND is present, the result is of type KIND.

## CHAR

Returns the character in the specified collating sequence.

F77

## Synopsis

```
CHAR(I [,KIND])
```

## Arguments

The argument I is of type integer, specifying the character position to return. The argument KIND is optional.

## Return Value

A character.

## CMPLX

Convert the supplied argument or arguments to complex type.

F77

## Synopsis

```
CMPLX(X [,Y][,KIND])
```

## Arguments

The argument X is of type integer, real, or complex. The optional argument Y is of type integer or real. If X is complex, Y must not be present. The optional argument KIND is the kind for the return value.

## Return Value

A complex number with the value specified by the arguments converted to a real part and a complex part.



If the `KIND` parameter is not supplied, the `KIND` is the same as the `KIND` for the default complex.

## COMPL

Performs a logical complement on the argument.

F77 extension

Synopsis

```
COMPL ( M )
```

Arguments

The argument `M` may be of any type except for character and complex.

Return Value

The return value is typeless.

## COMMAND\_ARGUMENT\_COUNT

Returns the number of command arguments.

F2003

Synopsis

```
COMMAND_ARGUMENT_COUNT ( )
```

Arguments

None.

Return Value

Returns a scalar of type default integer that is equal to the number of arguments passed on the command line when the containing program was invoked. If there were no command arguments passed, the result is 0.

Note

---

The command name does not count as one of the command arguments.

Example

```
program test_command_argument_count
  integer :: count
  count = command_argument_count()
  print *, count
end program test_command_argument_count
```

## CONJG

Return the conjugate of the supplied complex number.

COS

F77

## Synopsis

`CONJG ( Z )`

## Argument

The argument Z is a complex number.

## Return Value

A value of the same type and kind as the argument.

## COS

Return the cosine of the specified value.

F77

## Synopsis

`COS ( X )`

## Argument

The argument X must be of type real or complex.

## Return Value

A real value of the same kind as the argument. The return value for a real argument is in radians, or if complex, the real part is a value in radians.

## COSD

Return the cosine (in degrees) of the specified value.

F77

## Synopsis

`COSD ( X )`

## Argument

The argument X must be of type real or complex.

## Return Value

A real value of the same kind as the argument. The return value for a real argument is in degrees, or if complex, the real part is a value in degrees.

## COSH

Return the hyperbolic cosine of the specified value.

F77

### Synopsis

```
COSH ( X )
```

### Argument

The argument X must be of type real.

### Return Value

A real value.

## COUNT

Return the number of true elements in the supplied logical argument (array), along the specified dimension if the optional argument is present.

F90

### Synopsis

```
COUNT ( MASK [ , DIM ] )
```

### Arguments

The argument MASK is an array of type LOGICAL. The optional argument DIM specifies the dimension of the array MASK to count.

### Return Value

If no DIM argument is present, the return value is an integer that is the count of true values in MASK.

- If the DIM argument is present and if MASK has rank one, then the return value is the same as COUNT(MASK).
- If the DIM argument is present and MASK has rank greater than one, then the return value is an array that has rank n-1, where n is the rank of MASK. The return value is defined recursively as the value of COUNT for each extent of the dimension DIM (refer to The Fortran 95 Handbook for a more detailed explanation).

## CPU\_TIME

A non-elemental intrinsic subroutine that returns the processor time. For a more detailed explanation, refer to Fortran 95 Explained.

## CSHIFT

F95

### Synopsis

```
call cpu_time (TIME)
```

### Arguments

The argument `TIME` is a scalar real that is assigned a processor-dependent approximation of processor time.

### Return Value

The returned value in seconds, or a processor-dependent value if there is no clock.

## CSHIFT

Perform a circular shift on the specified array.

F90

### Synopsis

```
CSHIFT (ARRAY, SHIFT [, DIM])
```

### Arguments

The argument `ARRAY` is the array to shift. It may be an array of any type. The argument `SHIFT` is an integer or an array of integers with rank `n-1` where `n` is the rank of `ARRAY`. The optional argument `DIM` is an integer representing the dimension to shift.

### Return Value

The shifted array with the same size and shape as the argument `ARRAY`.

## DATE\_AND\_TIME

A subroutine that returns the date and time.

F90

### Synopsis

```
DATE_AND_TIME ([DATE] [, TIME] [, ZONE] [, VALUES])
```

### Arguments

All of the arguments are optional. The `DATE` argument is of type default character. It must be at least 8 characters long. The argument returns the value `CCYYMMDD` where `CC` is the century, `YY` is the year, `MM` is the month, and `DD` is the day.

The argument `TIME` is of type default character. It must be at least 10 characters long. It has the form `hhmmss.sss`, where `hh` is the hour, `mm` is the minute, and `ss.sss` is the seconds and milliseconds.

The argument `ZONE` is of type default character. It must be at least 5 characters long. It has the form `+-hhmm` where `hh` and `mm` are the hours and minutes that the local time zone differs from universal time (UTC).

The argument `VALUES` must be an array of type default integer. It has the following eight values:

```
VALUES(1) holds the year
VALUES(2) holds the month
VALUES(3) holds the day of the month
VALUES(4) holds the time difference with respect to UTC
VALUES(5) holds the hour of the day
VALUES(6) holds the minutes of the hour
VALUES(7) holds the seconds of the minute
VALUES(8) holds the milliseconds of the second,
           in the range 0 to 999
```

## Return Value

As this is a subroutine, the values are returned in the arguments.

## DBLE

Convert to double precision real.

F77

## Synopsis

```
DBLE(A)
```

## Argument

The argument `A` must be of type integer, real, or complex.

## Return Value

If `A` is of type integer or real, the return value is the value converted to a double precision real.

If `A` is of type complex, the return value is the double precision value of the real part of the complex argument.

## DCMPLX

Convert the supplied argument or arguments to double complex type.

F77

## Synopsis

```
DCMPLX(X [, Y])
```

## Arguments

The argument `X` is of type integer, real, or complex. The optional argument `Y` is of type integer or real. If `X` is complex, `Y` must not be present.

## Return Value

Returns a double complex number with the value specified by the arguments converted to a real part and a complex part.

## DIGITS

Returns the number of significant digits in the model representing the argument.

## F90

## Synopsis

```
DIGITS(X)
```

## Argument

The argument X is of type integer or real.

## Return Value

An integer value representing the number of digits in the model representing the specified kind.

## DIM

Returns the difference X-Y if the value is positive, otherwise it returns 0.

## F77

## Synopsis

```
DIM(X, Y)
```

## Arguments

X must be of type integer or real. Y must be of the same type and kind as X.

## Return Value

The result is the same type and kind as X with the value X-Y if X>Y, otherwise zero.

## DOT\_PRODUCT

Perform a dot product on two vectors (arrays).

## F90

## Synopsis

```
DOT_PRODUCT(VECTOR_A, VECTOR_B)
```

## Arguments

VECTOR\_A must be an array of rank one of type numeric (integer, real, complex) or logical. VECTOR\_B must be numeric if VECTOR\_A is numeric, or logical if VECTOR\_A is logical. It must have the same rank and size as ARRAY\_A.

## Return Value

The dot product.

- For VECTOR\_A of integer or real, the value is `SUM(VECTOR_A * VECTOR_B)`.
- For complex, the value is `SUM(CONJG(VECTOR_A) * VECTOR_B)`.
- For logical, the value is `ANY(VECTOR_A .AND. VECTOR_B)`.

## DPROD

Double precision real product.

## F90

## Synopsis

```
DPROD ( X, Y )
```

## Arguments

Both arguments X and Y must be of type default real.

## Return Value

A double precision real that is the product of X and Y.

## EOSHIFT

Perform an end-off shift on the specified array.

## F90

## Synopsis

```
EOSHIFT ( ARRAY, SHIFT [ , BOUNDARY ] [ , DIM ] )
```

## Arguments

The argument ARRAY is the array to shift. It may be an array of any type. The argument SHIFT is an integer or an array of integers with rank n-1 where n is the rank of ARRAY. The optional argument BOUNDARY is of the same type as the array, it may be scalar or of rank n-1 where n is the rank of ARRAY. The optional argument BOUNDARY is the value to fill in the shifted out positions. By default it has the following values for integer 0, for real 0.0, for complex (0.0,0.0), for logical false, for character the default is blank characters.

The optional argument DIM represents the dimension of ARRAY to shift.

## Return Value

The shifted array with the same size and shape as the argument `ARRAY`.

## EPSILON

Return the smallest number representable in the kind of the supplied argument.

## F90

### Synopsis

```
EPSILON(X)
```

### Argument

The argument `X` must be of type real.

### Return Value

A very small number in the specified real kind.

## EQV

Performs a logical exclusive NOR on the arguments.

## F77 extension

### Synopsis

```
EQV(M, N)
```

### Arguments

The arguments `M` and `N` may be of any type except for character and complex.

### Return Value

The return value is typeless.

## EXP

Exponential function.

## F77

### Synopsis

```
EXP(X)
```

### Argument

The argument `X` must be of type real or complex.



## Return Value

A value of the same type as the argument. It has the value `ex`.

## EXPONENT

Return the exponent part of a real number.

F90

## Synopsis

```
EXPONENT ( X )
```

## Argument

The argument `X` is a real number.

## Return Value

An integer which has the value of the exponent part of the value of `X`.

- If the exponent is zero, the function returns zero.
- If the exponent is too large to be defined as an integer, the result is undefined.

## EXTENDS\_TYPE\_OF

Determines whether the dynamic type of `A` is an extension type of the dynamic type of `B`.

F2003

## Synopsis

```
EXTENDS_TYPE_OF ( A , B )
```

## Argument

The arguments `A` and `B` must be objects of extensible type. If `A` or `B` is a pointer, it must not have an undefined association status.

## Return Value

A logical value which is based on the arguments `A` and `B`:

- If `B` is unlimited polymorphic and is a disassociated pointer or unallocated allocatable, the result is `TRUE`.
- If `A` is unlimited polymorphic and is either a disassociated pointer or unallocated allocatable, the result is `FALSE`.
- If the dynamic type of `A` is an extension type of the dynamic type of `B`, the result is `TRUE`.
- If none of the above conditions exist, the result is `FALSE`.

## FLOOR

Return the greatest integer less than or equal to the supplied real argument.

F90

### Synopsis

```
FLOOR(A [, KIND])
```

### Argument

The argument A is a real value. The optional argument KIND is a kind parameter and was added to FLOOR in Fortran 95.

### Return Value

An integer.

If KIND is present, the result is of type KIND.

## FRACTION

Return the fractional part of a real number.

F90

### Synopsis

```
FRACTION(X)
```

### Argument

The argument X is a real number.

### Return Value

The return value is an integer which has the value of the fractional part of the value of X. If the fraction value is zero, the function returns zero.

## GET\_COMMAND\_ARGUMENT

Returns the specified command line argument of the command that invoked the program.

F2003

### Synopsis

```
GET_COMMAND_ARGUMENT(NUMBER [, VALUE, LENGTH, STATUS])
```

### Arguments

The argument NUMBER indicates the position of the command-line argument to retrieve. NUMBER must be a scalar of type default integer. Further, NUMBER must be a non-negative value less than or equal to the value

returned by the function `COMMAND_ARGUMENT_COUNT`. If `NUMBER` is zero, then `VALUE` is set to the name by which the program was invoked.

The optional argument `VALUE`, a scalar of type default character, is the value of the returned argument, or blank if the value is unknown. If `VALUE` cannot hold the argument, the argument is truncated to fit.

The optional argument `LENGTH`, a scalar of type default integer, specifies the length of the returned command-line argument, or zero if the length of the argument is unknown. This length includes significant trailing blanks, but does not include any truncation or padding that occurs when the argument is assigned to the value argument.

The optional argument `STATUS`, a scalar of type default integer, specifies the returned completion status. A value of -1 indicates the value argument contains a truncated value for the command argument specified by `NUMBER`. A positive value indicates the argument retrieval failed. A value of `ZERO` indicates retrieval of the command argument was successful.

## Return Value

The return value is the specified command line argument of the command that invoked the program.

This subroutine cannot be passed as an actual argument.

## Example

```

program test_get_command_argument
  INTEGER :: i
  CHARACTER(len=32) :: arg
  i = 0
  DO
    CALL get_command_argument(i, arg)
    IF (LEN_TRIM(arg) == 0) EXIT
    WRITE (*,*) TRIM(arg)
    i = i+1
  END DO
END PROGRAM

```

## GET\_COMMAND

Returns the entire command line that was used to invoke the program.

F2003

## Synopsis

```
GET_COMMAND ([COMMAND, LENGTH, STATUS])
```

## Arguments

The optional argument *command*, a scalar of type default character, is the command that invoked the program, or a string of blanks if the command is unknown. If *command* cannot hold the entire command line, the command line is truncated to fit.

The optional argument *length*, a scalar of type default integer, specifies the significant length of the returned command that invoked the program, or zero if the length of the command is unknown. This length includes

## GET\_ENVIRONMENT\_VARIABLE

significant trailing blanks, but does not include any truncation or padding that occurs when the command is assigned to the *command* argument.

The optional argument *status*, a scalar of type default integer, specifies the returned completion status. A value of -1 indicates the *command* argument contains a truncated value for the command line retrieved. A value of 1 indicates the command retrieval failed. A value of ZERO indicates retrieval of the command was successful.

### Return Value

The return value is the specified command line that invoked the program.

### Example

```
PROGRAM test_get_command
  CHARACTER(len=255) :: cmd
  CALL get_command(cmd)
  WRITE (*,*) TRIM(cmd)
END PROGRAM test_get_command
```

## GET\_ENVIRONMENT\_VARIABLE

Returns the value of the specified environment variable.

### F2003

### Synopsis

```
GET_ENVIRONMENT_VARIABLE (NAME [,VALUE, LENGTH, STATUS, TRIM_NAME])
```

### Arguments

The argument *NAME*, a scalar of type character, indicates the name of the environment variable to retrieve.

The optional argument *VALUE*, a scalar of type default character, is the value of the returned environment variable, or blank if the environment variable does not exist or is not set. If *VALUE* cannot hold the argument, the argument is truncated to fit.

The optional argument *LENGTH*, a scalar of type default integer, specifies the length required to store the environment variable, or zero if the environment variable is not set. This length includes significant trailing blanks, but does not include any truncation or padding used in the command.

The optional argument *STATUS*, a scalar of type default integer, specifies the environment variable retrieval status. A value of -1 indicates the optional argument *VALUE* is too short to hold the environment variable. A positive value of 1 indicates the environment variable does not exist. A value of 2 indicated the processor does not support environment variables. A value of ZERO indicates the environment variable exists and either has no value or its value is successfully assigned to *VALUE*.

The optional argument *TRIM\_NAME*, a scalar of type logical. If *TRIM\_NAME* is present and has the value .FALSE., then the trailing blanks in *NAME* are significant; otherwise they are not part of the environment variable the value of the returned environment variable name.

## Return Value

The return value is the value of the specified environment variable.

## Example

```
PROGRAM test_get_environment_variable
  CHARACTER(len=255) :: homedir
  CALL get_environment_variable("HOME", homedir)
  WRITE (*,*) TRIM(homedir)
END PROGRAM test_get_environment_variable
```

## HUGE

Return the largest number representable in the kind of the supplied argument.

## F90

## Synopsis

```
HUGE(X)
```

## Argument

The argument X must be of type integer or real.

## Return Value

A value of the same type as the argument with the maximum value possible.

## IACHAR

Returns the position of the character in the ASCII collating sequence.

## F90

## Synopsis

```
IACHAR(C [,KIND])
```

## Argument

The argument C must be of type character. The optional KIND argument is a scalar integer initialization expression.

## Return Value

An integer representing the character position and of kind KIND. If KIND is absent, the return value is of default integer kind.

## Example

```
program test_iachar
  integer i
```

```

        i = iachar(' ')
    end program test_iachar

```

## IAND

Perform a bit-by-bit logical AND on the arguments.

F77

### Synopsis

```
IAND(I, J)
```

### Arguments

The arguments I and J must be of type integer of the same kind.

### Return Value

An integer value representing a bit-by-bit logical AND of the bits in the two integer arguments.

## IBCLR

Clears one bit to zero.

F77

### Synopsis

```
IBCLR(I, POS)
```

### Arguments

I is an integer. POS is a nonnegative integer less than BIT\_SIZE(I).

### Return Value

A value of the same type as I and with a value that is the same as I except that the bit in position POS is set to 0.

## IBITS

Extracts a sequence of bits.

F77

### Synopsis

```
IBITS(I, POS, LEN)
```

### Arguments

I is an integer. POS is a nonnegative integer and POS + LEN must be less than or equal to BIT\_SIZE(I). LEN is of type integer and is nonnegative.

## Return Value

A value that is the same type as I, with a value that is the sequence of LEN bits in I beginning at position POS, right-adjusted and with all other bits set to zero.

## IBSET

Set one bit to one.

F77

## Synopsis

```
IBSET(I, POS)
```

## Arguments

I is an integer. POS is a nonnegative integer less than BIT\_SIZE(I).

## Return Value

A value of the same type as I, with a value that is the same as I except that the bit in position POS is set to 1.

## ICHAR

Returns the position of a character in the character set's collating sequence.

F2003

## Synopsis

```
ICHAR(C [,KIND])
```

## Argument

The argument C must be of type character and length one. The optional KIND argument is a scalar integer initialization expression.

## Return Value

An integer representing the character position and of kind KIND. If KIND is absent, the return value is of default integer kind.

## Example

```
program test_ichar
  integer i
  i = ichar(' ')
end program test_ichar
```

## IEOR

Perform a bit-by-bit logical exclusive OR on the arguments.

## IINT

### F77

#### Synopsis

```
IEOR(I, J)
```

#### Argument

The arguments I and J must be of type integer of the same kind.

#### Return Value

An integer value representing a bit-by-bit logical Exclusive OR of the bits in the two integer arguments.

## IINT

Converts a value to a short integer type.

### F77 extension

#### Synopsis

```
IINT(A)
```

#### Arguments

The argument A is of type integer, real, or complex.

#### Return Value

The return value is the short integer value of the supplied argument. For a real number, if the absolute value of the real is less than 1, the return value is 0. If the absolute value is greater than 1, the result is the largest short integer that does not exceed the real value. If argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

## INDEX

Returns the starting position of a substring within a string.

### F2003

#### Synopsis

```
INDEX(STRING, SUBSTRING [,BACK[, KIND]])
```

#### Arguments

The argument STRING must be of type character string. The argument SUBSTRING must be of type character string with the same kind as STRING. The optional argument BACK must be of type logical. The optional KIND argument is a scalar integer initialization expression.



## Return Value

The function returns an integer of kind `KIND`. If `KIND` is absent, the return value is of default integer kind.

If `BACK` is absent or false, the result is the starting point of the first matching `SUBSTRING` within `STRING`. Zero is returned if no match is found. 1 is returned if the `SUBSTRING` has zero length.

If `BACK` is present with the value true, the result is the last matching substring in string, or zero if no match is found.

## ININT

Returns the nearest short integer to the real argument.

### F77 extension

### Synopsis

```
ININT ( A )
```

### Arguments

The argument `A` must be a real.

### Return Value

A short integer with value  $(A + .5 * \text{SIGN}(A))$ .

## INT

Converts a value to integer type.

### F77

### Synopsis

```
INT ( A [ , KIND ] )
```

### Arguments

The argument `A` is of type integer, real, or complex. The optional argument `KIND` must be a scalar integer that is a valid kind for the specified type. The `KIND` argument is not allowed by pgf77.

### Return Value

The integer value of the supplied argument.

- For a real number, if the absolute value of the real is less than 1, the return value is 0.
- If the absolute value is greater than 1, the result is the largest integer that does not exceed the real value.

- If the argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

## INT8

Converts a real value to a long integer type.

### F77 extension

#### Synopsis

```
INT8 ( A )
```

#### Arguments

The argument A is of type real.

#### Return Value

The long integer value of the supplied argument.

## IOR

Perform a bit-by-bit logical OR on the arguments.

### F77

#### Synopsis

```
IOR ( I , J )
```

#### Argument

The arguments I and J must be of type integer of the same kind.

#### Return Value

An integer value representing a bit-by-bit logical OR of the bits in the two integer arguments.

## ISHFT

Perform a logical shift.

### F77

#### Synopsis

```
ISHFT ( I , SHIFT )
```

#### Arguments

I and SHIFT are integer values. The absolute value of SHIFT must be less than or equal to BIT\_SIZE(I).

## Return Value

A value of the same type and kind as the argument `I`, the value of the argument `I` logically shifted by `SHIFT` bits. If `SHIFT` is positive, the shift is to the left. If `SHIFT` is negative, the shift is to the right. Zeroes are shifted in at the ends and the bits shifted out are lost.

## IS\_IOSTAT\_END

Test whether a variable has the value of the I/O status: “end of file”.

### Note

---

This function is equivalent to comparing the variable with the `IOSTAT_END` parameter of the intrinsic module `ISO_FORTRAN_ENV`.

## F2003

## Synopsis

```
IO_STAT_END(stat)
```

## Arguments

`stat` is an integer value.

## Return Value

A LOGICAL of the default kind:

- `.TRUE.` if `stat` has the value which indicates an end of file condition for `IOSTAT=` specifiers.
- `.FALSE.` if `stat` has any value other than one that indicates an end of file condition for `IOSTAT=` specifiers.

## Example

```
program iostat_test
  Implicit NONE
  integer :: endstat, i
  open(88, File='test.dat')
  read(88, *, IOSTAT=endstat) i
  If(IS_IOSTAT_END(endstat)) Stop "END OF FILE"
end program iostat_test
```

## IS\_IOSTAT\_EOR

Test whether a variable has the value of the I/O status: “end of record”.

### Note

---

This function is equivalent to comparing the variable with the `IOSTAT_EOR` parameter of the intrinsic module `ISO_FORTRAN_ENV`.

## F2003

## Synopsis

```
IO_STAT_EOR(stat)
```

## Arguments

`stat` is an integer value.

## Return Value

A LOGICAL of the default kind:

- `.TRUE.` if `stat` has the value which indicates an end of record condition for `IOSTAT=` specifiers.
- `.FALSE.` if `stat` has any value other than one that indicates an end of record condition for `IOSTAT=` specifiers.

## Example

```
program iostat_test_eor
  Implicit NONE
  integer :: endstat, i
  open(88, File='test.dat', FORM='UNFORMATTED')
  read(88, *, IOSTAT=endstat) i
  If (IS_IOSTAT_EOR(endstat)) Stop "END OF RECORD"
end program iostat_test_eor
```

## ISHFTC

Perform a circular shift of the rightmost bits.

## F77

## Synopsis

```
ISHFTC(I, SHIFT [,SIZE])
```

## Arguments

`I` and `SHIFT` are integer values. The absolute value of `SHIFT` must be less than or equal to the optional argument `SIZE`. If present, `SIZE` must not exceed the value `BIT_SIZE(I)`; if `SIZE` is not present, the function acts as if it were present with the value `BIT_SIZE(I)`.

## Return Value

The value of the sub-group of `SIZE` bits shifted by `SHIFT` positions within the sub-group, with all other bits remaining the same and in the same position.

- If the `SIZE` argument is not present, the circular shift is over the complete group of bits and all of the bits are shifted by `SHIFT` positions.
- If `SHIFT` is positive, the shift is to the left. If `SHIFT` is negative, the shift is to the right.

## IZEXT

Zero-extend the argument.

F77 extension

Synopsis

```
IZEXT(A)
```

Arguments

The argument A is of type logical or integer.

Return Value

A zero-extended short integer of the argument.

## JINT

Converts a value to an integer type.

F77 extension

Synopsis

```
JINT(A)
```

Arguments

The argument A is of type integer, real, or complex.

Return Value

The integer value of the supplied argument.

- For a real number, if the absolute value of the real is less than 1, the return value is 0.
- If the absolute value is greater than 1, the result is the largest integer that does not exceed the real value.
- If argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

## JNINT

Returns the nearest integer to the real argument.

F77 extension

Synopsis

```
JNINT(A)
```

KIND

## Arguments

The argument A must be a real.

## Return Value

An integer with value  $(A + .5 * \text{SIGN}(A))$ .

## KIND

Returns the kind of the supplied argument.

F90

## Synopsis

```
KIND ( X )
```

## Argument

The argument X is of any intrinsic type.

## Return Value

An integer representing the kind type parameter of X.

## KNINT

Returns the nearest integer to the real argument.

F77 extension

## Synopsis

```
KNINT ( A )
```

## Arguments

The argument A must be a real.

## Return Value

A long integer with value  $(A + .5 * \text{SIGN}(A))$ .

## LBOUND

Returns the lower bounds of an array, or the lower bound for the specified dimension.

F2003

## Synopsis

```
LBOUND ( ARRAY [ , DIM [ , KIND ] ] )
```

## Arguments

The argument `ARRAY` is an array of any type. The optional argument `DIM` is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and  $n$  where  $n$  is the rank of the array). The optional `KIND` argument is a scalar integer initialization expression.

## Return Value

An integer of kind `KIND`, or an array of kind `KIND` with rank one and size  $n$ , where  $n$  is the rank of the argument `ARRAY`. If `ARRAY` is an expression rather than a whole array or array structure component, or if it has a zero extent along the relevant dimension, the lower bound is taken to be 1.

- For the function with a `DIM` argument, the return value is the value of the lower bound in the specified dimension.
- For the function with no `DIM` supplied, the return value is an array with all the lower bounds of `ARRAY`.
- If `KIND` is absent, the return value is of default integer kind.

## LEADZ

Counts the number of leading zero bits.

F2003

## Synopsis

```
LEADZ ( I )
```

## Arguments

`I` is of type integer or bits.

## Return Value

The result is one of the following:

- If all of the bits of `I` are zero: `BIT SIZE (I)`.
- If at least one of the bits of `I` is not zero: `BIT SIZE (I) - 1 - k`.

`k` is the position of the leftmost 1 bit in `I`.

## Description

`LEADZ` is an elemental function that returns the number of leading zero bits.

## Examples

The following example returns the value 2.

## LEN

```
LEADZ (B'001101000')
```

The following example returns the value 31 if BIT SIZE (1) has the value 32.

```
LEADZ (1)
```

## LEN

Returns the length of the supplied string.

## F2003

### Synopsis

```
LEN (STRING[ , KIND ])
```

### Argument

The argument **STRING** is a character string or an array. The optional **KIND** argument is a scalar integer initialization expression.

### Return Value

An integer of kind **KIND** that represents the length of the scalar string supplied, or the length of an element of **STRING** if **STRING** is an array. If **KIND** is absent, the return value is of default integer kind.

## LEN\_TRIM

Returns the length of the supplied string minus the number of trailing blanks.

## F2003

### Synopsis

```
LEN_TRIM (STRING[ , KIND ])
```

### Arguments

The argument **STRING** is a character string.

### Return Value

An integer of kind **KIND** that represents the length of the scalar string minus the number of trailing blanks, if any. If **KIND** is absent, the return value is of default integer kind.

## LGE

Test the supplied strings to determine if the first string **STRING\_A** is lexically greater than or equal to the second string **STRING\_B**. A string is lexically greater than another string if the first string follows the second string alphabetically.



F90

## Synopsis

```
LGE (STRING_A, STRING_B)
```

## Argument

The arguments `STRING_A` and `STRING_B` are of type default character.

## Return Value

A logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

## LGT

Test the supplied strings to see if the first string `STRING_A` is lexically greater than the second string `STRING_B`. A string is lexically greater than another string if the first string follows the second string alphabetically.

F90

## Synopsis

```
LGT (STRING_A, STRING_B)
```

## Argument

The arguments `STRING_A` and `STRING_B` are of type default character.

## Return Value

A logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

## LLE

Test the supplied strings to see if the first string `STRING_A` is lexically less than or equal to the second string `STRING_B`. A string is lexically less than another string if the first string precedes the second string alphabetically.

F90

## Synopsis

```
LLE (STRING_A, STRING_B)
```

## Argument

The arguments `STRING_A` and `STRING_B` are of type default character.

LLT

## Return Value

A logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

## LLT

Test the supplied strings to see if the first string `STRING_A` is lexically less than the second string, `STRING_B`. A string is lexically less than another string if the first string precedes the second string alphabetically.

F90

## Synopsis

```
LLT(STRING_A, STRING_B)
```

## Argument

The arguments `STRING_A` and `STRING_B` are of type default character.

## Return Value

A logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

## LOC

Return the 32-bit address of a data item.

F77

## Synopsis

```
LOC(X)
```

## Argument

The argument `X` is of type integer, real or complex.

## Return Value

An integer representing the address of the argument.

## LOG

A function that returns the natural logarithm.

F77

## Synopsis

```
LOG(X)
```

200

## Argument

The argument *X* is of type real or complex. If *X* is real, it must be greater than 0. If *X* is complex, it must not be equal to zero.

## Return Value

The natural log (base *e*) of *X*.

## LOG10

A function that returns the common logarithm.

## F77

## Synopsis

```
LOG10(X)
```

## Actions:LOG;Argument

The argument *X* is of type real and must be greater than 0.

## Return Value

The common log (base 10) of *X*.

## LOGICAL

Convert a logical value to the specified logical kind.

## F90

## Synopsis

```
LOGICAL(L [,KIND])
```

## Arguments

The argument *L* is the logical value to convert. The optional argument *KIND* must be a scalar integer that is a valid kind for the specified type.

## Return Value

A logical value equal to the logical value *L*.

If *KIND* is specified, the kind type parameter of the return value is that of *KIND*, otherwise it is default logical.

## LSHIFT

Perform a logical shift to the left.

## F77 extension

## Synopsis

```
LSHIFT(I, SHIFT)
```

## Arguments

I and SHIFT are integer values.

## Return Value

A value of the same type and kind as the argument I. It is the value of the argument I logically shifted left by SHIFT bits.

**MATMUL**

Perform a matrix multiply of numeric or logical matrices.

## F90

## Synopsis

```
MATMUL(MATRIX_A, MATRIX_B)
```

## Arguments

The argument `MATRIX_A` must be numeric (integer, real, or complex) or logical, and have a rank of one or two. The argument `MATRIX_B` must be numeric (integer, real, or complex) or logical, and have a rank of one or two. If `MATRIX_A` has rank one, then `MATRIX_B` must have rank 2. If `MATRIX_B` has rank one, then `MATRIX_A` must have rank 2. The size of the first dimension of `MATRIX_B` must equal the size of the last dimension of `MATRIX_A`.

## Return Value

A matrix representing the value of the matrix multiplied arguments. There are three possible result shapes:

- `MATRIX_A(n,m)` and `MATRIX_B(m,k)` gives a result (n,k) matrix.
- `MATRIX_A(m)` and `MATRIX_B(m,k)` gives a result (k).
- `MATRIX_A(n,m)` and `MATRIX_B(m)` gives a result(n).

**MAX**

Return the maximum value of the supplied arguments.

## F77

## Synopsis

```
MAX( A1, A2 [,A3,...])
```

## Arguments

Any arguments after the first two are optional. The arguments must all have the same kind and they must be integer or real.

## Return Value

A value the same as the type and kind of the arguments, having the value of the largest argument.

## MAXEXPONENT

Returns the value of the maximum exponent for the type and the kind supplied.

## F90

## Synopsis

```
MAXEXPONENT ( X )
```

## Argument

The argument X must be a scalar or an array of type real.

## Return Value

An integer whose value is that of the largest exponent in the specified kind.

## MAXLOC

Determine the first position in the specified array that has the maximum value of the values in the array. The test elements may be limited by a dimension argument or by a logical mask argument.

## F2003

## Synopsis

```
MAXLOC ( ARRAY [ , DIM ] [ , MASK ] [ , KIND ] )
```

## Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM, added in Fortran 95, is of type integer. The optional argument MASK must be of type logical and must have the same shape as ARRAY. If only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK. The optional KIND argument is a scalar integer initialization expression.

## Return Value

An integer array of rank 1 with size equal to the number of dimensions in ARRAY.

- The return values represent the positions of the first element in each dimension that is the maximum value of that dimension.

- If the MASK parameter is present, the return value is the position of the first value that has the maximum value of values in ARRAY, and that also has a true value in the corresponding MASK array.
- When the DIM argument is supplied, the return value is an array that has a value of MAXLOC applied recursively along the DIM dimensions of the array.
- If KIND is absent, the return value is of default integer kind.

## MAXVAL

Return the maximum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

F90

### Synopsis

```
MAXVAL(ARRAY [ , DIM] [ , MASK ])
```

### Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and n where n is the rank of the array). The optional argument MASK must be of type logical and must have the same shape as ARRAY. Fortran 95 has extended MAXVAL such that if only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

### Return Value

Varies, based on the arguments supplied:

- The return value is a scalar if no DIM argument is present, or has a rank of n-1 and a shape specified by all of the dimensions except the DIM argument dimension.
- The return value is the value of the largest element of the array if no optional parameters are supplied.
- If only the MASK parameter is supplied with the array, then the return value is the value that is the maximum of the true elements of MASK.
- When the DIM argument is supplied, the return value is an array that has a value of MAXVAL applied recursively along the DIM dimensions of the array.

## MERGE

A function that merges two arguments based on the value of a logical mask.

F90

### Synopsis

```
MERGE(TSOURCE, FSOURCE, MASK)
```

## Arguments

TSOURCE is the source that is merged if the mask is true. FSOURCE is the source that is merged if the mask is false. TSOURCE and FSOURCE must be of the same type and must have the same type parameters (if they are arrays they must be conformable). MASK must be of type logical.

## Return Value

A value with the same type and type parameters as the source arrays.

## MIN

Return the minimum value of the supplied arguments.

## F77

## Synopsis

```
MIN(A1, A2 [,A3,...])
```

## Arguments

Any arguments after the first two are optional. The arguments must all have the same kind and they must be integer or real.

## Return Value

The same as the type and kind of the arguments, with the value of the smallest argument.

## MINEXPONENT

Returns the value of the minimum exponent for the type and the kind supplied.

## F90

## Synopsis

```
MINEXPONENT(X)
```

## Argument

The argument X has type real and may be a scalar or an array.

## Return Value

An integer that has the value of the smallest exponent in the specified kind.

## MINLOC

Return the position of the element with the minimum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

## MINVAL

F90

### Synopsis

```
MINLOC(ARRAY [ ,DIM] [ ,MASK] [ ,KIND])
```

### Arguments

The argument `ARRAY` must be an array of type integer or real. The optional argument `DIM`, added in Fortran 95, is of type integer. The optional argument `MASK` must be of type logical and must have the same shape as `ARRAY`. If only two arguments are supplied, the type of the second argument is used to determine if it represents `DIM` or `MASK`. The optional `KIND` argument is a scalar integer initialization expression.

### Return Value

An integer array of rank 1 with a size equal to the number of dimensions in `ARRAY`. The value is the position of the first element that is the minimum value of the array.

- If the `MASK` parameter is present, the return value is the position of the first value that has the minimum value of values in `ARRAY`, and that also has a true value in the corresponding `MASK` array.
- When the `DIM` argument is supplied, the return value is an array that has a value of `MINLOC` applied recursively along the `DIM` dimensions of the array.
- If `KIND` is absent, the return value is of default integer kind.

## MINVAL

Return the minimum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

F90

### Synopsis

```
MINVAL(ARRAY [ ,DIM] [ ,MASK])
```

### Arguments

The argument `ARRAY` must be an array of type integer or real. The optional argument `DIM` is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and `n` where `n` is the rank of the array). The optional argument `MASK` must be of type logical and must have the same shape as `ARRAY`. Fortran 95 has extended `MINVAL` such that if only two arguments are supplied, the type of the second argument is used to determine if it represents `DIM` or `MASK`.

### Return Value

Varies depending on the arguments supplied:

- A scalar if no `DIM` argument is present, or a value with a rank of `n-1` and a shape specified by all of the dimensions except the `DIM` argument dimension.



- The return value is the value of the largest element of the array if no optional parameters are supplied.
- If only the MASK parameter is supplied with the array, then the return value is the value that is the minimum of the true elements of MASK.
- When the DIM argument is supplied, the return value is an array that has a value of MINVAL applied recursively along the DIM dimensions of the array.

## MOD

Find the remainder.

F77

### Synopsis

```
MOD(A, P)
```

### Arguments

The argument A must be an integer or a real. The argument P must be of the same type and kind as A.

### Return Value

A value of the same type as the argument A and value  $(A - \text{INT}(A/P) * P)$ .

## MODULO

Return the modulo value of the arguments.

F90

### Synopsis

```
MODULO(A, P)
```

### Arguments

The argument A must be an integer or a real. The argument P must be of the same type and kind as A.

### Return Value

The return value is the same type as the argument A.

- If A and P are of type real, the result is  $(A - \text{FLOOR}(A/P) * P)$ .
- If A and P are of type integer, the result is  $(A - \text{FLOOR}(A \div P) * P)$  where  $\div$  represents ordinary mathematical division.

## MOVE\_ALLOC

Moves an allocation from one allocatable object to another.

## F2003

## Synopsis

```
MOVE_ALLOC (FROM, TO)
```

## Arguments

The FROM argument is allocatable and of any type. The TO argument is allocatable and of the same type and rank as the FROM argument.

## Return Value

Move\_alloc is a subroutine and has no return value; instead, after the call, the allocation status and target, if any, of the TO argument is modified to that of the FROM argument before the call. Further, the FROM argument becomes deallocated.

## Example

```
REAL,ALLOCATABLE :: a(:),tmp(:)
...
ALLOCATE(a(n))
...
! Double the size of A without losing any of the values already stored
ALLOCATE(tmp(size(a)*2))
tmp(1:size(a)) = a
CALL MOVE_ALLOC(from=tmp,to=a)
! TMP is deallocated; A has the new size and values.
```

## MVBITS

Copies a bit sequence from a source data object to a destination data object.

## F90

## Synopsis

```
MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)
```

## Arguments

All arguments have type integer. The arguments FROMPOS, LEN, and TOPOS must be nonnegative. The TO argument must be a variable of type integer and have the same kind type parameter as the FROM argument.

## Return Value

MVBITS is a subroutine and has no return value; instead, the TO argument is modified. LEN bits starting at FROMPOS in FROM are copied to TO at TOPOS. All other bits of TO remain unchanged.

## NEAREST

Returns the nearest different machine representable number in a given direction.

F90

## Synopsis

```
NEAREST ( X, S )
```

## Arguments

The argument X is a real number. The argument S is a real number and not equal to zero.

## Return Value

A value of the same type as X that contains the value that is the closest possible different machine representable number from X in the direction given by the sign of S.

## NEQV

Performs a logical exclusive OR on the arguments.

## F77 extension

## Synopsis

```
COMPL ( M, N )
```

## Arguments

The arguments M and N may be of any type except for character and complex.

## Return Value

The return value is typeless.

## NEW\_LINE

Return the newline character.

F2003

## Synopsis

```
NEW_LINE ( A )
```

## Arguments

The argument A must be of type character and specifies the kind of the result.

## Return Value

The newline character.

## NINT

Returns the nearest integer to the real argument.

F77

### Synopsis

```
NINT(A [ , KIND ])
```

### Arguments

The argument A must be a real. The optional argument KIND specifies the kind of the result integer.

### Return Value

An integer.

- If  $A > 0$ , NINT(A) has the value is INT(A+0.5).
- If A is less than or equal to 0, NINT(A) has the value INT(A-0.5).

## NOT

Perform a bit-by-bit logical complement on the argument.

F77

### Synopsis

```
NOT(I)
```

### Argument

The argument I must be of type integer.

### Return Value

An integer value representing a bit-by-bit logical complement of the bits in the argument.

## NULL

Gives the disassociated status to pointer entities. Fortran 95 added this transformational function. For a more detailed explanation, refer to Fortran 95 Explained.

F95

### Synopsis

```
NULL([ PTR ])
```

## Arguments

The optional argument PTR is a pointer of any type and may have any association status including undefined.

## Return Value

A disassociated pointer.

## OR

Performs a logical OR on each bit of the arguments.

## F77 extension

## Synopsis

```
OR ( M , N )
```

## Arguments

The arguments M and N may be of any type except for character and complex.

## Return Value

The return value is typeless.

## PACK

Pack an array of any number of dimensions into an array of rank one.

## F90

## Synopsis

```
PACK ( ARRAY , MASK [ , VECTOR ] )
```

## Arguments

The ARRAY argument is the array to be packed and may be of any type. The MASK argument is of type logical and must be conformable with ARRAY. The optional argument VECTOR is of the same type as ARRAY and has rank one.

## Return Value

A packed array limited by the logical values in the array MASK. If VECTOR is present its values are part of the result array only for those elements that have an element order greater than the number of true elements in MASK. For further details, refer to the Fortran 95 Handbook.

## POPCNT

Return the number of one bits.

## F2008

## Synopsis

```
POPCNT ( I )
```

## Arguments

I is of type integer or bits.

## Integer

The result value is equal to the number of one bits in the sequence of bits of I.

If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of the default integer type.

## Bits

The result value is the number of one bits in I.

## Return Value

The number of one bits.

## Description

POPCNT is an elemental function that returns the number of one bits

## Examples

The following example returns the value [1, 1, 2, 1, 2, 2].

```
POPCNT ([1, 2, 3, 4, 5, 6])
```

The following example returns the value 16.

```
POPCNT (Z'FFFF0000')
```

The following example has the same value as KIND (B).

```
POPCNT (HUGE (B))
```

**POPPAR**

Return the bitwise parity.

## F2008

## Synopsis

```
POPPAR ( I )
```

## Arguments

I is of type integer or bits.

## Return Value

An integer

- 1 if POPCNT(I) is odd
- 0 if POPCNT(I) is even.

## Description

POPPAR is an elemental function that returns the bitwise parity of the argument.

## Examples

The following example returns the value [1, 1, 0, 1, 0, 0].

```
POPPAR ([1, 2, 3, 4, 5, 6])
```

The following example returns the value 0.

```
POPPAR (Z'FFFF0000')
```

## PRECISION

Return the decimal precision of the real or complex argument.

### F90

## Synopsis

```
PRECISION(X)
```

## Argument

The argument X must be a real or complex number.

## Return Value

An integer representing the decimal precision of the argument.

## PRESENT

Determine if an optional argument is present.

### F90

## Synopsis

```
PRESENT(A)
```

## Argument

The argument A must be an optional argument in the procedure in which the intrinsic is called.

## PRODUCT

### Return Value

A logical scalar. True if A is present and false otherwise.

## PRODUCT

Returns the product of the elements of the supplied array.

F90

### Synopsis

```
PRODUCT ( ARRAY [ , DIM ] [ , MASK ] )
```

### Arguments

The **ARRAY** argument is an array of integer, real or complex type. The optional **DIM** argument is a valid dimension (valid dimensions are between the values 1 and n where n is the rank of the array). The optional **MASK** argument is of type logical and conformable with the supplied array. Fortran 95 has extended **PRODUCT** such that if only two arguments are supplied, the type of the second argument is used to determine if it represents **DIM** or **MASK**.

### Return Value

The product of the elements of **ARRAY**.

- If the optional **DIM** argument is present, the product is for the specified dimension.
- If the optional **MASK** argument is present, the result is subject to the logical mask supplied.

## RADIX

Return the base of the model representing numbers of the type and kind of the argument.

F90

### Synopsis

```
RADIX ( X )
```

### Argument

The argument **X** is of type integer or real.

### Return Value

An integer with the value of the radix (base) of the number system model of the argument.

## RANDOM\_NUMBER

Returns one pseudorandom number or an array of pseudo-random numbers from the uniform distribution over the range  $0 \leq x < 1$ .



F90

## Synopsis

```
RANDOM_NUMBER ( HARVEST )
```

## Argument

The argument HARVEST must of type real. It is set to contain the resulting pseudorandom number or array of pseudorandom numbers from the uniform distribution.

## Return Value

RANDOM\_NUMBER is a subroutine.

## Description

The random number intrinsic generates a 46 bit lagged fibonacci pseudo-random sequence with a short lag of 5 and a long lag of 17. For a given seed, including the default seed, the sequence generated is independent of the platform and number of processors. Due to limitations of some platforms' default integer type, the seed vector is of size 34. Only the least significant 23 bits of each element of the seed array are used, thus a seed array returned or used is portable between platforms. For non-degenerate seed arrays, the period of this generator is  $(2^{17} - 1) * 2^{45}$ . If all the odd elements of the seed array are even, the period will be shorter.

For the PGHPF compiler, the best performance on distributed arrays is for block distributions. The higher the order of the first distributed dimension, the better the performance will be.

**RANDOM\_SEED**

Restarts or queries the pseudorandom number generator for RANDOM\_NUMBER.

F90

## Synopsis

```
RANDOM_SEED( [ SIZE ] [ , PUT ] [ , GET ] )
```

## Arguments

The arguments SIZE, PUT and GET are optional. There must be one or no arguments. Multiple arguments are not allowed. SIZE is an integer value representing the number of integers that the processor uses to hold the value of the seed. PUT is an integer array of rank one and is used to set the seed. GET is an integer array of rank one and is used to get the value of the seed.

## Return Value

RANDOM\_SEED is a subroutine.

**RANGE**

Return the decimal exponent range for the type of number supplied as an argument.

REAL

F90

## Synopsis

```
RANGE ( X )
```

## Argument

The argument X must be of type integer, real, or complex.

## Return Value

An integer.

## REAL

Convert the argument to real.

F77

## Synopsis

```
REAL ( A [ , KIND ] )
```

## Arguments

The argument A must be of type integer, real, or complex. The optional argument KIND specifies the kind type of the result.

## Return Value

A real number. For a complex argument, the imaginary part is ignored.

## REPEAT

Concatenate copies of a string.

F90

## Synopsis

```
REPEAT ( STRING , NCOPIES )
```

## Arguments

The argument STRING must be a scalar of type character. The argument NCOPIES is an integer.

## Return Value

A character string that is NCOPIES times as long as STRING. It is the concatenation of STRING NCOPIES times.

## RESHAPE

Reconstructs an array with the specified shape using the elements of the source array.

F90

### Synopsis

```
RESHAPE(SOURCE, SHAPE [, PAD] [, ORDER])
```

### Arguments

The argument `SOURCE` is an array of any type. The argument `SHAPE` is of type integer and has rank one. It must not have more than 7 elements and no values can be negative. The optional argument `PAD` must be the same size and type as `SOURCE`. The optional argument `ORDER` must be of type integer and must have the same shape as `SHAPE`.

### Return Value

An array of shape `SHAPE`, with the same type as `SOURCE`. Array elements are filled into the new array in array element order.

## RRSPACING

Return the reciprocal of the relative spacing of model numbers near the argument value.

F90

### Synopsis

```
RRSPACING(X)
```

### Argument

The argument `X` is of type real.

### Return Value

A value of the same type as `X`.

## RSHIFT

Perform a logical shift to the right.

F77 extension

### Synopsis

```
RSHIFT(I, SHIFT)
```

## SAME\_TYPE\_AS

### Arguments

I and SHIFT are integer values.

### Return Value

A value of the same type and kind as the argument I. It is the value of the argument I logically shifted right by SHIFT bits.

## SAME\_TYPE\_AS

Determines whether the dynamic type of A is the same as the dynamic type of B.

### F2003

### Synopsis

```
SAME_TYPE_AS ( A , B )
```

### Argument

The arguments A and B must be objects of extensible type. If A or B is a pointer, it must not have an undefined association status.

### Return Value

A logical scalar of TRUE if and only if the dynamic type of A is the same as the dynamic type of B. Otherwise, returns FALSE.

## SCALE

Return the value  $X * b^i$  where b is the base of the number system in use for X.

### F90

### Synopsis

```
SCALE ( X , I )
```

### Arguments

The argument X is of type real. The argument I is an integer.

### Return Value

A real value of the same type as the argument X.

## SCAN

Search the supplied string for a character in a set of characters.

F90

## Synopsis

```
SCAN(STRING, SET [,BACK[,KIND]])
```

## Arguments

The argument `STRING` is of type character and is the string to search. The argument `SET` is of type character and has the same kind type parameter as `STRING`. The optional argument `BACK` is of type logical. The optional `KIND` argument is a scalar integer initialization expression.

## Return Value

An integer of type `KIND`, specifying the position in `STRING` of a character from `SET`. If the optional parameter `BACK` is not present, or is present and false, the result is the position of the first character found. If `BACK` is present and true, the return value is that of the last character in `STRING` matching one in `SET`. If `KIND` is absent, the return value is of default integer kind.

## Example

```
PROGRAM test_scan
  WRITE(*,*) SCAN("FORTRAN", "AO")           ! 2, found 'O'
  WRITE(*,*) SCAN("FORTRAN", "AO", .TRUE.)  ! 6, found 'A'
  WRITE(*,*) SCAN("FORTRAN", "C++")        ! 0, found none
END PROGRAM
```

**SELECTED\_INT\_KIND**

Returns a value that is the kind type parameter that will represent a number in the specified range, where the range is determined by the formula  $-10R < n < 10R$ , where  $n$  is an integer and  $R$  is the argument.

F90

## Synopsis

```
SELECTED_INT_KIND(R)
```

## Argument

The argument `R` must be of type integer.

## Return Value

An integer. If the value `R` is invalid, the return value is `-1`.

**SELECTED\_REAL\_KIND**

Returns a value that is the kind type parameter that will represent a number in the specified range, where the range is determined with decimal precision `P` and a decimal exponent range of at least `R`.

## SET\_EXPONENT

F90

### Synopsis

```
SELECTED_REAL_KIND([P] [,R])
```

### Arguments

The arguments are both optional, but at least one of the optional arguments must be present. The argument P must be of type integer and specifies a precision. The argument R must be of type integer and specifies a range.

### Return Value

An integer.

## SET\_EXPONENT

Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

F90

### Synopsis

```
SET_EXPONENT(X, I)
```

### Arguments

The argument X is of type real. The argument I is of type integer.

### Return Value

A value of the type of the argument X.

## SHAPE

Returns the shape of the supplied argument.

F90

### Synopsis

```
SHAPE(SOURCE[,KIND])
```

### Arguments

The argument SOURCE is a scalar or an array of any type. The optional KIND argument is a scalar integer initialization expression.

### Return Value

An array whose size is equal to the rank of SOURCE and whose values represent the shape of SOURCE. If KIND is absent, the return value is of default integer kind.

## Example

```
PROGRAM test_shape
  INTEGER, DIMENSION(-1:1, -1:2) :: A
  WRITE(*,*) SHAPE(A)           ! (/ 3, 4 /)
  WRITE(*,*) SIZE(SHAPE(42))   ! (/ /)
END PROGRAM
```

## SHIFT

Perform a logical shift.

### F77 extension

### Synopsis

```
RSHIFT(I, SHIFT)
```

### Arguments

The argument *I* may be of any type except character or complex. The argument *SHIFT* is of type integer.

### Return Value

The return value is typeless. If *SHIFT* is positive, the result is *I* logically shifted left by *SHIFT* bits. If *SHIFT* is negative, the result is *I* logically shifted right by *SHIFT* bits.

## SIGN

Return the absolute value of *A* times the sign of *B*.

### F77

### Synopsis

```
SIGN(A, B)
```

### Arguments

The argument *A* is an integer or real number. The argument *B* must be of the same type as *A*.

### Return Value

The value of the absolute value of *A* times the sign of *B*, with the same type as *A*. If *B* is zero, its sign is taken as positive.

#### Note

---

Fortran 95 allows for a distinction to be made between positive and negative real zeroes. In this case, if *B* is a real zero, its sign is positive if it is a positive real zero or if the processor cannot distinguish between positive and negative real zeroes.

SIN

## SIN

Return the value of the sine of the argument.

F90

### Synopsis

```
SIN ( X )
```

### Argument

The argument X must be of type real or complex.

### Return Value

A value that has the same type as X and is expressed in radians.

## SIND

Return the value in degrees of the sine of the argument.

F77

### Synopsis

```
SIND ( X )
```

### Argument

The argument X must be of type real or complex.

### Return Value

A value that has the same type as X and is expressed in degrees.

## SINH

Return the hyperbolic sine of the argument.

F77

### Synopsis

```
SINH ( X )
```

### Argument

The argument X must be of type real.

### Return Value

A value that has the same type as X.



## SIZE

Returns either the total number of elements in the array or the number of elements along a specified dimension.

F2003

### Synopsis

```
SIZE (ARRAY [ ,DIM][ ,KIND])
```

### Arguments

The argument `ARRAY` is an array of any type. The optional `DIM` argument must be a valid dimension (valid dimensions are between the values 1 and `n` where `n` is the rank of the array). The optional `KIND` argument is a scalar integer initialization expression.

### Return Value

The result value is an integer of kind `KIND`. If `DIM` is absent, the function returns the total number of elements in the array. If `DIM` is present, the function returns the extent of the array in the specified dimension. If `KIND` is absent, the return value is of default integer kind.

### Examples

```
PROGRAM test_size
  WRITE(*,*) SIZE((/ 1, 2 /))    ! 2
END PROGRAM
```

## SPACING

Returns the spacing of model numbers near the argument.

F90

### Synopsis

```
SPACING(X)
```

### Arguments

The argument `X` is of type real.

### Return Value

A value that has the same type and kind as `X`.

## SPREAD

Replicates an array by adding a new dimension.

## SQRT

F90

### Synopsis

```
SPREAD(SOURCE, DIM, NCOPIES)
```

### Arguments

The argument `SOURCE` may be of any type with rank less than 7. The `DIM` argument is a scalar integer representing a valid dimension (valid dimensions are between the values 1 and `n` where `n` is the rank of the array). The argument `NCOPIES` must be scalar and of type integer.

### Return Value

An array of the same type as the `SOURCE`, with rank `n+1` where `n` is the rank of `SOURCE`.

## SQRT

Return the square root of the argument.

F77

### Synopsis

```
SQRT(X)
```

### Arguments

The argument `X` must be a real or complex number.

### Return Value

The value of the same type as the argument.

## SUM

Returns the sum of the elements of the supplied array.

F90

### Synopsis

```
SUM(ARRAY [,DIM] [,MASK])
```

### Arguments

The `ARRAY` argument is an array of integer, real or complex type. The optional `DIM` argument is a valid dimension (valid dimensions are between the values 1 and `n` where `n` is the rank of the array). The optional `MASK` argument is of type logical and conformable with the supplied array. Fortran 95 has extended `SUM` such that if only two arguments are supplied, the type of the second argument is used to determine if it represents `DIM` or `MASK`.

## Return Value

The sum of the elements of the argument `ARRAY`.

- If the optional `DIM` argument is present, the sum is for the specified dimension.
- If the optional `MASK` argument is present, the result is subject to the logical mask supplied.

## SYSTEM\_CLOCK

Returns information about the real time clock.

F90

### Synopsis

```
SYSTEM_CLOCK ( [ COUNT ] [ , COUNT_RATE ] [ , COUNT_MAX ] )
```

### Arguments

- `COUNT`, an optional argument, is a scalar integer that provides the current count of the system clock when the subroutine is called.
- `COUNT_RATE`, an optional argument, is an integer or a real value that provides the number of clock ticks per second.

The number of ticks per second is always 1000. This routine is implemented on most systems using `gettimeofday(2)`; some implementations use `dclock(3)`.

- `COUNT_MAX`, an optional argument, is a scalar integer that provides the value of the maximum count possible.

## Return Value

The arguments of this subroutine are modified during the call; there is no return value.

## TAN

Return the tangent of the specified value.

F77

### Synopsis

```
TAN ( X )
```

### Argument

The argument `X` must be of type real and have absolute value  $\leq 1$ .

## Return Value

A real value of the same `KIND` as the argument.

## TAND

Return the tangent of the specified value.

F77

### Synopsis

```
TAND ( X )
```

### Argument

The argument X must be of type real and have absolute value  $\leq 1$ .

### Return Value

A real value of the same KIND as the argument.

## TANH

Return the hyperbolic tangent of the specified value.

F77

### Synopsis

```
TANH ( X )
```

### Argument

The argument X must be of type real and have absolute value  $\leq 1$ .

### Return Value

A real value of the same KIND as the argument.

## TINY

Return the smallest positive number representable in the kind of the supplied argument.

F90

### Synopsis

```
TINY ( X )
```

### Argument

The argument X must be of type real.

### Return Value

The smallest positive number in the number system. The value has the same type as the argument X.

## TRANPOSE

Transpose an array of rank two.

F90

### Synopsis

```
TRANPOSE ( MATRIX )
```

### Arguments

The argument `MATRIX` is a two-dimensional array of any type.

### Return Value

A transformed matrix with the same type as `MATRIX` and dimensions  $(m,n)$  where matrix `MATRIX` has dimensions  $(n,m)$ .

## TRIM

Remove the trailing blanks from a string.

F90

### Synopsis

```
TRIM ( STRING )
```

### Arguments

The argument `STRING` is the string to be adjusted and must be a scalar.

### Return Value

The same as the argument but with the trailing blanks removed. The size of the returned string is the size of the argument `STRING` minus the number of trailing blanks in `STRING`.

## UBOUND

Returns the upper bounds of an array or the upper bound for the specified dimension.

F90

### Synopsis

```
UBOUND ( ARRAY [ , DIM ] [ , KIND ] )
```

### Arguments

The argument `ARRAY` is an array of any type. The optional argument `DIM` is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and  $n$  where  $n$  is the rank of the array). The optional `KIND` argument is a scalar integer initialization expression.

## Return Value

An integer of kind `KIND` or an array of kind `KIND` with rank one and size `n`, where `n` is the rank of the argument `ARRAY`.

- If `DIM` is not supplied, the return value is an array with all the upper bounds for `ARRAY`.
- If `DIM` is provided, the return value is the value of the upper bound in the specified dimension.
- If `KIND` is absent, the return value is of default integer kind.

## UNPACK

Unpack an array of rank one dimension into an array of any number of dimensions.

### F90

#### Synopsis

```
UNPACK(VECTOR, MASK, FIELD)
```

#### Arguments

The `VECTOR` argument is an array of any type and of rank one. It must have as many elements as there are true elements in `MASK`. The `MASK` argument is of type logical array. The `FIELD` argument must be the same type as `VECTOR` and must be conformable with `MASK`.

#### Return Value

An array that has the same type as `VECTOR` and the shape of `MASK`. For further details and information on the `FIELD` argument, refer to The Fortran 95 Handbook.

## VERIFY

Verify that a character string contains all characters from a set of characters.

### F2003

#### Synopsis

```
VERIFY(STRING, SET [,BACK[,KIND]])
```

#### Arguments

The arguments `STRING` and `SET` are of type character. The optional argument `BACK` is of type logical. `BACK` is a logical that determines if the first or last character position is returned. The optional `KIND` argument is a scalar integer initialization expression.

#### Return Value

An integer of kind `KIND`. The function returns the position of the first (or last) character that is not in the set.

- If BACK is present and true, the position of the right-most character is returned.
- If BACK is not present or present and false, the position of the left-most character is returned.
- If KIND is absent, the return value is of default integer kind.

## Example

```
PROGRAM test_verify
  WRITE(*,*) VERIFY("FORTRAN", "AO")           ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "FOO")          ! 3, found 'R'
  WRITE(*,*) VERIFY("FORTRAN", "C++")         ! 1, found 'F'
  WRITE(*,*) VERIFY("FORTRAN", "C++", .TRUE.) ! 7, found 'N'
  WRITE(*,*) VERIFY("FORTRAN", "FORTRAN")     ! 0' found none
END PROGRAM
```

## XOR

Performs a logical exclusive OR on each bit of the arguments.

### F77 extension

### Synopsis

```
XOR(M, N)
```

### Arguments

The arguments M and N must be of integer type.

### Return Value

An integer.

## ZEXT

Zero-extend the argument.

### F77 extension

### Synopsis

```
ZEXT(A)
```

### Arguments

The argument A is of type logical or integer.

### Return Value

An integer.

## Intrinsic Modules

Like an intrinsic function, the Fortran processor provides the intrinsic module. It is possible for a program to use an intrinsic module and a user-defined module of the same name, though they cannot both be referenced from the same scope.

- To use a user-defined module rather than an intrinsic module, specify the keyword *non-intrinsic* in the USE statement:

```
USE, non-intrinsic :: iso_fortran_env
```

- To use an intrinsic module rather than a user-defined one, specify the keyword *intrinsic* in the USE statement:

```
USE, intrinsic :: iso_fortran_env
```

### Note

---

If both a user-defined and intrinsic module of the same name are available and locatable by the compiler, a USE statement without either of the keywords previously described accesses the user-defined module. If the compiler cannot locate the user-defined module, it accessed the intrinsic module and does not issue a warning.

## Module IEEE\_ARITHMETIC

The `ieee_arithmetic` intrinsic module provides access to two derived types, named constants of these types, and a collection of generic procedures.

### Note

---

This module behaves as if it contained a **use** statement for the module `ieee_exceptions`, so all the features of `ieee_exceptions` are included. For information of this module, refer to [“Module IEEE\\_EXCEPTIONS,” on page 234](#).

## IEEE\_ARITHMETIC Derived Types

The `ieee_arithmetic` intrinsic module provides access to these two derived types: `ieee_class_type` and `ieee_round_type`.

`ieee_class_type`

Identifies a class of floating point values.

`ieee_round_type`

Identifies a particular round mode.

For both of these types, the following are true:

- The components are private.
- The only operations defined are `==` and `/=`.
- The return value is of type default logical.
- Intrinsic assignment is available.



Table 6.15 provides a quick overview of the values that each derived type can take.

Table 6.15. IEEE\_ARITHMETIC Derived Types

This derived type...	Must have one of these values...
ieee_class_type	ieee_signaling_nan ieee_quiet_nan ieee_negative_inf ieee_negative_normal ieee_negative_denormal ieee_negative_zero ieee_positive_zero ieee_positive_denormal ieee_positive_normal ieee_positive_inf ieee_other_value (Fortran 2003 only)
ieee_round_type	ieee_nearest - ieee_to_zero ieee_up ieee_down ieee_other (for modes other than IEEE ones)

### IEEE\_ARITHMETIC Inquiry Functions

The `ieee_arithmetic` intrinsic module supports a number of inquiry functions. Table 6.16 provides a list and brief description of what it means if the inquiry function returns `.true.`. In all cases, if the condition for returning `.true.` is not met, the function returns `.false.`.

Table 6.16. IEEE\_ARITHMETIC Inquiry Functions

	Returns <code>.true.</code> if ...	
This inquiry function...	When optional arg. <code>x</code> is absent	When optional arg. <code>x</code> is present
<code>ieee_support_datatype([x])</code>	The processor supports IEEE arithmetic for all reals	The processor supports IEEE arithmetic for all reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_denormal([x])</code>	The processor supports IEEE denormalized numbers for all reals	The processor supports IEEE denormalized numbers for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_divide([x])</code>	The processor supports divide with the accuracy specified by IEEE standard for all reals	The processor supports divide with the accuracy specified by IEEE standard for reals of the same kind type parameter as the real argument <code>x</code> .

	Returns <code>.true.</code> if ...	
This inquiry function...	When optional arg. <code>x</code> is absent	When optional arg. <code>x</code> is present
<code>ieee_support_inf([x])</code>	The processor supports the IEEE infinity facility for all reals	The processor supports the IEEE infinity facility for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_nan([x])</code>	The processor supports the IEEE Not-A-Number facility for all reals	The processor supports the IEEE Not-A-Number facility for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_rounding(round_value[,x])</code>	For a <code>round_value</code> of <code>ieee_round_type</code> , the processor supports the rounding mode numbers for all reals	For a <code>round_value</code> of <code>ieee_round_type</code> , the processor supports the rounding mode numbers for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_sqrt([x])</code>	The processor implements the IEEE square root for all reals	The processor implements the IEEE square root for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_standard([x])</code>	The processor supports all IEEE facilities for all reals	The processor supports all IEEE facilities for reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_underflow_control([x])</code>	(Fortran 2003 only) The processor supports control of the underflow mode for all reals	(Fortran 2003 only) The processor supports control of the underflow mode for reals of the same kind type parameter as the real argument <code>x</code> .

### IEEE\_ARITHMETIC Elemental Functions

The `ieee_arithmetic` intrinsic module supports a number of elemental functions. [Table 6.17](#) provides a list and brief description of the return value. In all cases involving a return value of true or false, if the condition for returning `.true.` is not met, the subroutine returns `.false.`

Table 6.17. IEEE\_ARITHMETIC Elemental Functions

This elemental function...	Does this...
<code>ieee_class(x)</code>	Returns the IEEE class of the real argument <code>x</code> .
<code>ieee_copy_sign(x,y)</code>	Returns a real with the same type parameter as the real argument <code>x</code> , holding the value of <code>x</code> with the sign of <code>y</code> .

This elemental function...	Does this...
ieee_is_finite(x)	Returns <code>.true.</code> if <code>ieee_class(x)</code> has one of these values: <code>ieee_negative_normal</code> <code>ieee_negative_denormal</code> <code>ieee_negative_zero</code> <code>ieee_positive_zero</code> <code>ieee_positive_denormal</code> <code>ieee_positive_normal</code>
ieee_is_nan(x)	Returns <code>.true.</code> if the value of <code>x</code> is an IEEE NaN.
ieee_is_negative(x)	Returns <code>.true.</code> if <code>ieee_class(x)</code> as one of these values: <code>ieee_negative_normal</code> <code>ieee_negative_denormal</code> <code>ieee_negative_zero</code> <code>ieee_negative_inf</code>
ieee_is_normal(x)	Returns <code>.true.</code> if <code>ieee_class(x)</code> has one of these values: <code>ieee_negative_normal</code> <code>ieee_negative_zero</code> <code>ieee_positive_zero</code> <code>ieee_positive_normal</code>
ieee_is_logb(x)	Returns a real with the same type parameter as the argument <code>x</code> . If <code>x</code> is neither zero, infinity, nor NaN, the value of the result is the unbiased exponent of <code>x</code> : <code>exponent(x) - 1</code> . If <code>x</code> is 0 and <code>ieee_support_inf(x)</code> is true, the result is <code>-infinity</code> . If <code>x</code> is 0 and <code>ieee_support_inf(x)</code> is not true, the result is <code>-huge(x)</code> .
ieee_next_after(x,y)	Returns a real with the same type parameter as the argument <code>x</code> . If <code>x == y</code> , the result is <code>x</code> . Otherwise, the result is the neighbor of <code>x</code> in the direction of <code>y</code> .
ieee_rem(x,y)	Returns a real with the same type parameter of whichever argument has the greater precision.
ieee_rint(x,y)	Returns a real with the same type parameter as <code>x</code> , and whose value is that of <code>x</code> rounded to an integer value according to the current rounding mode.
ieee_scalb(x,i)	Returns a real with the same type parameter as <code>x</code> , and whose value is $2^i x$ . If $2^i x$ is too large, <code>ieee_overflow</code> signals. If $2^i x$ is too small, <code>ieee_underflow</code> signals.
ieee_unordered(x,y)	Returns <code>.true.</code> if <code>x</code> or <code>y</code> or both are a NaN.
ieee_value(x,class)	Returns a real with the same type parameter as <code>x</code> and a value specified by <code>class</code> .

## IEEE\_ARITHMETIC Non-Elemental Subroutines

The `ieee_arithmetic` intrinsic module supports a number of elemental functions. Table 6.17 provides a list and brief description of what it means if the inquiry function returns `.true.` In all cases, if the condition for returning `.true.` is not met, the function returns `.false.`

In these subroutines, the argument `round_value` is a scalar of type `ieee_round` type and the argument `gradual` is a scalar of type default `logical`.

Table 6.18. IEEE\_ARITHMETIC Non-Elemental Subroutines

This non-elemental subroutine...	Does this...
<code>ieee_get_rounding_mode(round_value)</code>	Returns the floating-point rounding mode. If one of the IEEE modes is in operation, the value is one of these: <code>ieee_nearest</code> <code>ieee_to_zero</code> <code>ieee_up</code> <code>ieee_down</code> Otherwise, the value is <code>ieee_positive_normal</code>
<code>ieee_get_underflow_mode(gradual)</code>	Returns <code>.true.</code> if gradual underflow is in effect-point rounding mode. Otherwise, it returns <code>.false.</code>
<code>ieee_set_rounding_mode(round_value)</code>	Specifies the rounding mode to be set.
<code>ieee_set_underflow_mode(gradual)</code>	Sets gradual underflow in effect if the value is <code>.true.</code> ; otherwise, gradual underflow ceases to be in effect.

## IEEE\_ARITHMETIC Transformational Function

The `ieee_arithmetic` intrinsic module supports `ieee_selected_real_kind([p] [,r])` a transformational function that is permitted in an initialization expression.

This result of this function is the kind value of a real `x` for which `ieee_support_datatype(x)` is `true.`

## Module IEEE\_EXCEPTIONS

The `ieee_exceptions` intrinsic module provides support for overflow and divide by zero flags in the scoping unit for all available kinds of reals and complex data. This module contains two derived types, named constants of these types, and a collection of generic procedures.

### IEEE\_EXCEPTIONS Derived Types

- **`ieee_flag_type`** - Identifies a particular exception flag.
- **`ieee_status_type`** - Saves the current floating-point status.

For both of these types, the following are true:

- The components are private.

- No operations are defined for these types.
- Only intrinsic assignment is available.

Table 6.19 provides a quick overview of the values that each derived type can take.

Table 6.19. IEEE\_EXCEPTIONS Derived Types

This derived type...	Must have one of these values...
<code>ieee_flag_type</code>	For named constants: <code>ieee_overflow</code> <code>ieee_underflow</code> <code>ieee_divide_by_zero</code> <code>ieee_inexact</code> <code>ieee_invalid</code>
<code>ieee_status_type</code>	Includes the values of all supported flags as well as current rounding mode.

### IEEE\_EXCEPTIONS Inquiry Functions

The `ieee_exceptions` intrinsic module supports two inquiry functions, both of which are pure:

- `ieee_support_flag(flag [,x])`
- `ieee_support_halting(flag)`

For both functions, the argument `flag` must be of type `type(ieee_flag_type)`.

Table 6.20 provides a list and brief description of what it means if the inquiry function returns `.true.` In all cases, if the condition for returning `.true.` is not met, the function returns `.false..`

Table 6.20. IEEE\_EXCEPTIONS Inquiry Functions

This inquiry function...	Returns <code>.true.</code> if ...
<code>ieee_support_flag(flag [,x])</code>	The processor supports the exception <code>flag</code> for all reals. If the optional argument <code>x</code> is present, then it returns <code>.true.</code> if the processor supports the exception <code>flag</code> for all reals of the same kind type parameter as the real argument <code>x</code> .
<code>ieee_support_halting(flag)</code>	The processor supports the ability to change the mode by call <code>ieee_set_halting(flag)</code> .

### IEEE\_EXCEPTIONS Subroutines Functions

The `ieee_exceptions` intrinsic module supports elemental and non-elemental subroutines.

In all these subroutines:

- `flag` is of type `type(ieee_flag_type)`
- `halting` is of type `default logical`

- `flag_value` is of type `default logical`
- `status_value` if is type `type(ieee_status_type)`.

## Elemental Subroutines

Table 6.21 provides a list and brief description of what it means if the inquiry function returns `.true.` In all cases, if the condition for returning `.true.` is not met, the function returns `.false.`

Table 6.21. IEEE\_EXCEPTIONS Elemental Subroutines

This elemental subroutine...	Does this...
<code>ieee_get_flag(flag, flag_value)</code>	If the value of <code>flag</code> is <code>ieee_invalid</code> , <code>ieee_overflow</code> , <code>ieee_divide_by_zero</code> , <code>ieee_underflow</code> , or <code>ieee_inexact</code> and the corresponding exception flag is signaling, <code>flag_value</code> is <code>true</code> .
<code>ieee_get_halting_mode(flag, halting)</code>	The value <code>flag</code> must have one of the values: <code>ieee_invalid</code> , <code>ieee_overflow</code> , <code>ieee_divide_by_zero</code> , <code>ieee_underflow</code> , or <code>ieee_inexact</code> . If the exception specified causes halting, <code>halting</code> is <code>true</code> .

## Non-Elemental Subroutines

The `ieee_exceptions` intrinsic module supports non-elemental subroutines for flags and halting mode as well as for floating-point status.

Table 6.21 provides a list and brief description of these subroutines.

Table 6.22. IEEE\_EXCEPTIONS Elemental Subroutines

This non-elemental subroutine...	Does this...
<code>ieee_set_flag(flag, flag_value)</code>	If the value returned by <code>ieee_support_halting</code> is <code>true</code> , each <code>flag</code> specified is set to be signalling if the corresponding <code>flag_value</code> is <code>true</code> and is set to be quiet if it is <code>false</code> .
<code>ieee_get_halting_mode(flag, halting)</code>	Each exception specified by <code>flag</code> causes halting if the corresponding value of <code>halting</code> is <code>true</code> . If value is <code>false</code> , it does not cause halting.
<code>ieee_get_status(status_value)</code>	Returns the floating-point status, including all the exception flags, the rounding mode, and the halting mode.
<code>ieee_set_status(status_value)</code>	Resets the floating-point status, including all the exception flags, the rounding mode, and the halting mode to the previous invocation of <code>ieee_get_status</code> .

## Module `iso_c_binding`

The `iso_c_binding` intrinsic module provides access to named constants of type `default integer` that represent kind type parameters of data representations compatible with C types.

- A positive value indicates that the Fortran type and kind type parameter interoperate with the corresponding C type.
- A negative value indicates a lack of support.

## Module `iso_fortran_env`

The `iso_fortran_env` intrinsic module provides information about the Fortran environment through named constants. The following table provides the constants and a brief description of the information provided. Each named constant is a default integer scalar.

Table 6.23. `iso_fortran_env` Named Constants

<b>This Named Constant...</b>	<b>Provides this Fortran environment information...</b>
<code>character_storage_size</code>	The size, in bits, of a character storage unit
<code>error_unit</code>	The unit number for a preconnected output unit suitable for reporting errors.
<code>file_storage_size</code>	The size, in bits, of a file storage unit.
<code>input_unit</code>	The unit number for the preconnected external unit used for input.
<code>iostat_end</code>	The value returned by <code>IOSTAT=</code> that indicates an end-of-file condition occurs during execution of a <code>READ</code> statement.
<code>iostat_eor</code>	The value returned by <code>IOSTAT=</code> that indicates an end-of-record condition occurs during execution of a <code>READ</code> statement.
<code>numeric_storage_size</code>	The size, in bits, of a numeric storage unit.
<code>output_unit</code>	The unit number for the preconnected external unit used for output.

These special unit numbers may be negative, though they are never -1, since -1 is reserved for another purpose.

The error-unit may be the same as output-unit.





# Chapter 7. 3F Functions and VAX Subroutines

The PGI Fortran compilers support FORTRAN 77 3F functions and VAX/VMS system subroutines and built-in functions.

## 3F Routines

This section describes the functions and subroutines in the Fortran run-time library which are known as 3F routines on many systems. These routines provide an interface from Fortran programs to the system in the same manner as the C library does for C programs. These functions and subroutines are automatically loaded from PGI's Fortran run-time library if referenced by a Fortran program.

The implementation of many of the routines uses functions which reside in the C library. If a C library does not contain the necessary functions, undefined symbol errors will occur at link-time. For example, if PGI's C library is the C library available on the system, the following 3F routines exist in the Fortran run-time library, but use of these routines will result in errors at link-time:

besj0	besj1	besjn	besy0	besy1	besyn
dbesj0	dbesj1	dbesjn	dbesy0	dbesy1	dbesyn
derf	derfc	erf	erfc	getlog	hostnm
lstat	putenv	symlnk	ttynam		

The routines `mclock` and `times` depend on the existence of the C function `times()`.

The routines `dtime` and `etime` are only available in a `SYSVR4` environment. These routines are not available in all environments simply because there is no standard mechanism to resolve the resolution of the value returned by the `times()` function.

There are several 3F routines, such as `fputc` and `fgetc`, that perform I/O on a logical unit. These routines bypass normal Fortran I/O. If normal Fortran I/O is also performed on a logical unit which appears in any of these routines, the results are unpredictable.

## abort

Terminate abruptly and write memory image to core file.

### Synopsis

```
subroutine abort()
```

### Description

The `abort` function cleans up the I/O buffers and then aborts, producing a core file in the current directory.

## abort

Terminate abruptly and write memory image to core file.

### Synopsis

```
subroutine abort()
```

### Description

The `abort` function cleans up the I/O buffers and then aborts, producing a core file in the current directory.

## access

Determine access mode or existence of a file.

### Synopsis

```
integer function access(fil, mode)
character*(*) fil
character*(*) mode
```

### Description

The `access` function tests the file, whose name is `fil`, for accessibility or existence as determined by `mode`.

The `mode` argument may include, in any order and in any combination, one or more of:

- `r`  
test for read permission
- `w`  
test for write permission
- `x`  
test for execute permission
- (blank)  
test for existence

An error code is returned if either the `mode` argument is illegal or if the file cannot be accessed in all of the specified modes. Zero is returned if the specified access is successful.

## alarm

Execute a subroutine after a specified time.

### Synopsis

```
integer function alarm(time, proc)
integer time
external proc
```

### Description

This routine establishes subroutine `proc` to be called after `time` seconds. If `time` is 0, the alarm is turned off and no routine will be called. The return value of `alarm` is the time remaining on the last alarm.

## Bessel functions

These functions calculate Bessel functions of the first and second kinds for real and double precision arguments and integer orders.

```
besj0
besj1
besjn
besy0
besy1
besyn
dbesj0
dbesj1
dbesjn
dbesy0
dbesy1
dbesyn
```

### Synopsis

```
real function besj0(x)
real x
real function besj1(x)
real x
```

```
real function besjn(n, x)
integer n
real x
real function besy0(x)
real x
real function besy1(x)
real x
real function besyn(n, x)
integer n
real x
double precision function dbesj0(x)
double precision x
double precision function dbesj1(x)
double precision x
double precision function dbesjn(n, x)
integer n
double precision x
double precision function dbesy0(x)
```

```
double precision x
double precision function dbesyl(x)
double precision x
double precision function dbesyn(n, x)
integer n
double precision x
```

## chdir

Change default directory.

### Synopsis

```
integer function chdir(path)
character*(*) path
```

### Description

Change the default directory for creating and locating files to `path`. Zero is returned if successful; otherwise, an error code is returned.

## chmod

Change mode of a file.

### Synopsis

```
integer function chmod(nam, mode)
character*(*) nam
integer mode
```

### Description

Change the file system mode of file `nam`. If successful, a value of 0 is returned; otherwise, an error code is returned.

## ctime

Return the system time.

### Synopsis

```
character*(*) function ctime(stime)
integer*8 stime
```

### Description

`ctime` converts a system time in `stime` to its ASCII form and returns the converted form. Neither newline nor NULL is included.

## date

Return the date.

## Synopsis

```
character*(*) function date(buf)
```

## Description

Returns the ASCII representation of the current date. The form returned is dd-mmm-yy.

## error functions

The functions erf and derf return the error function of x. erfc and derfc return 1.0-erf(x) and 1.0-derf(x), respectively.

## Synopsis

```
real function erf(x)
real x
real function erfc(x)
real x
double precision function derf(x)
double precision x
double precision function derfc(x)
double precision x
```

## etime, dtime

Get the elapsed time.

## Synopsis

```
real function etime(tarray)
real function dtime(tarray)
real tarray(2)
```

## Description

etime returns the total processor run-time in seconds for the calling process.

dtime (delta time) returns the processor time since the previous call to dtime. The first time it is called, it returns the processor time since the start of execution.

Both functions place values in the argument tarray: user time in the first element and system time in the second element. The return value is the sum of these two times.

## exit

Terminate program with status.

## Synopsis

```
subroutine exit(s)
integer s
```

fdate

## Description

exit flushes and closes all of the program's files, and returns the value of s to the parent process.

## fdate

Return date and time in ASCII form.

## Synopsis

```
character*(*) function fdate()
```

## Description

fdate returns the current date and time as a character string. Neither newline nor NULL will be included.

## fgetc

Get character from a logical unit.

## Synopsis

```
integer function fgetc(lu, ch)  
integer lu  
character*(*) ch
```

## Description

Returns the next character in ch from the file connected to the logical unit lu, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

## flush

Flush a logical unit.

## Synopsis

```
subroutine flush(lu)  
integer lu
```

## Description

flush flushes the contents of the buffer associated with logical unit lu.

## fork

Fork a process.

## Synopsis

```
integer function fork()
```

## Description

fork creates a copy of the calling process. The value returned to the parent process will be the process id of the copy. The value returned to the child process (the copy) will be zero. If the returned value is negative, an error occurred and the value is the negation of the system error code.

## fputc

Write a character to a logical unit.

## Synopsis

```
integer function fputc(lu, ch)
integer lu
character*(*) ch
```

## Description

A character ch is written to the file connected to logical unit lu bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

## free

Free memory.

## Synopsis

```
subroutine free(p)
int p
```

## Description

Free a pointer to a block of memory located by malloc; the value of the argument, p, is the pointer to the block of memory.

## fseek

Position file at offset.

## Synopsis

```
integer function fseek(lu, offset, from)
integer lu
integer offset
integer from
```

## Description

fseek repositions a file connected to logical unit lu. offset is an offset in bytes relative to the position specified by from :

0  
beginning of the file

ftell

- 1  
current position
- 2  
end of the file

If successful, the value returned by fseek will be zero; otherwise, it's a system error code.

## ftell

Determine file position.

### Synopsis

```
integer function ftell(lu)
integer lu
```

### Description

ftell returns the current position of the file connected to the logical unit lu. The value returned is an offset, in units of bytes, from the beginning of the file. If the value returned is negative, it is the negation of the system error code.

## gerror

Return system error message.

### Synopsis

```
character*(*) function gerror()
```

### Description

Return the system error message of the last detected system error.

## getarg

Get the nth command line argument.

### Synopsis

```
subroutine getarg(n, arg)
integer n
character*(*) arg
```

### Description

Return the nth command line argument in arg, where the 0th argument is the command name.

## iargc

Index of the last command line argument.



## Synopsis

```
integer function iargc()
```

## Description

Return the index of the last command line argument, which is also the number of arguments after the command name.

## getc

Get character from unit 5.

## Synopsis

```
integer function getc(ch)
character*(*) ch
```

## Description

Returns the next character in ch from the file connected to the logical unit 5, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

## getcwd

Get pathname of current working directory.

## Synopsis

```
integer function getcwd(dir)
character*(*) dir
```

## Description

The pathname of the current working directory is returned in dir. If successful, the return value is zero; otherwise, an error code is returned.

## getenv

Get value of environment variable.

## Synopsis

```
subroutine getenv(en, ev)
character*(*) en
character*(*) ev
```

## Description

getenv checks for the existence of the environment variable en. If it does not exist or if its value is not present, ev is filled with blanks. Otherwise, the string value of en is returned in ev.

getgid

## getgid

Get group id.

### Synopsis

```
integer function getgid()
```

### Description

Return the group id of the user of the process.

## getlog

Get user's login name.

### Synopsis

```
character*(*) function getlog()
```

### Description

getlog returns the user's login name or blanks if the process is running detached from a terminal.

## getpid

Get process id.

### Synopsis

```
integer function getpid()
```

### Description

Return the process id of the current process.

## getuid

Get user id.

### Synopsis

```
integer function getuid()
```

### Description

Return the user id of the user of the process.

## gmtime

Return system time.

## Synopsis

```
subroutine gmtime(stime, tarray)
integer stime
integer tarray(9)
```

## Description

Dissect the UNIX time, `stime`, into month, day, etc., for GMT and return in `tarray`.

## hostnm

Get name of current host.

## Synopsis

```
integer function hostnm(nm)
character*(*) nm
```

## Description

`hostnm` returns the name of the current host in `nm`. If successful, a value of zero is returned; otherwise an error occurred.

## idate

Return date in numerical form.

## Synopsis

```
subroutine idate(im, id, iy)
integer im, id, iy
```

## Description

Returns the current date in the variables `im`, `id`, and `iy`, which indicate the month, day, and year, respectively. The month is in the range 1-12; only the last 2 digits of the year are returned.

## ierrno

Get error number.

## Synopsis

```
integer function ierrno()
```

## Description

Return the number of the last detected system error.

## ioinit

Initialize I/O

## Synopsis

```
subroutine ioinit(cctl, bzro, apnd, prefix, vrbose)
integer cctl
integer bzro
integer apnd
character*(*) prefix
integer vrbose
```

## Description

Currently, no action is performed.

## isatty

Is logical unit a tty.

## Synopsis

```
logical function isatty(lu)
integer lu
```

## Description

Returns *.TRUE.* if logical unit *lu* is connected to a terminal; otherwise, *.FALSE.* is returned.

## itime

Return time in numerical form.

## Synopsis

```
subroutine itime(iarray)
integer iarray(3)
```

## Description

Return current time in the array *iarray*. The order is hour, minute, and second.

## kill

Send signal to a process.

## Synopsis

```
integer function kill(pid, sig)
integer pid
integer sig
```

## Description

Send signal number *sig* to the process whose process id is *pid*. If successful, the value zero is returned; otherwise, an error code is returned.

## link

Make link

### Synopsis

```
integer function link(n1, n2)
character* (*) n1
character* (*) n2
```

### Description

Create a link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

## lnblnk

Return index of last non-blank.

### Synopsis

```
integer function lnblnk(a1)
character* (*) a1
```

### Description

Return the index of the last non-blank character in string a1.

## loc

Address of an object.

### Synopsis

```
integer function loc(a)
integer a
```

### Description

Return the value which is the address of a.

## ltime

Return system time.

### Synopsis

```
subroutine ltime(stime, tarray)
integer stime
integer tarray(9)
```

### Description

Dissect the UNIX time, stime , into month, day, etc., for the local time zone and return in tarray.

malloc

## malloc

Allocate memory.

### Synopsis

```
integer function malloc(n)
integer n
```

### Description

Allocate a block of n bytes of memory and return the pointer to the block of memory.

## mclock

Get elapsed time.

### Synopsis

```
integer function mclock()
```

### Description

mclock returns the sum of the user's cpu time and the user and system times of all child processes. The return value is in units of clock ticks per second.

## mvbits

Move bits.

### Synopsis

```
subroutine mvbits(src, pos, len, dest, posd)
integer src
integer pos
integer len
integer dest
integer posd
```

### Description

len bits are moved beginning at position pos of argument src to position posd of argument dest.

## outstr

Print a character string.

### Synopsis

```
integer function outstr(ch)
character*(*) ch
```

## Description

Output the character string to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error occurred.

## **perror**

Print error message.

## Synopsis

```
subroutine perror(str)
character*(*) str
```

## Description

Write the message indicated by str to logical unit 0 and the message for the last detected system error.

## **putc**

Write a character to logical unit 6.

## Synopsis

```
integer function putc(ch)
character*(*) ch
```

## Description

A character ch is written to the file connected to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

## **putenv**

Change or add environment variable.

## Synopsis

```
integer function putenv(str)
character*(*) str
```

## Description

str contains a character string of the form name=value. This function makes the value of the environment variable name equal to value. If successful, zero is returned.

## **qsort**

Quick sort.

## Synopsis

```
subroutine qsort(array, len, isize, compar)
```

```
dimension array(*)
integer len
integer isize
external compar
integer compar
```

## Description

qsort sorts the elements of the one dimensional array, array. len is the number of elements in the array and isize is the size of an element. compar is the name of an integer function that determines the sorting order. This function is called with 2 arguments (arg1 and arg2) which are elements of array. The function returns:

negative

if arg1 is considered to precede arg2

zero

if arg1 is equivalent to arg2

positive

if arg1 is considered to follow arg2

## rand, irand, srand

Random number generator.

## Synopsis

```
double precision function rand()
integer function irand()
subroutine srand(iseed)
integer iseed
```

## Description

The functions rand and irand generate successive pseudo-random integers or double precision numbers. srand uses its argument, iseed, to re-initialize the seed for successive invocations of rand and irand.

irand

returns a positive integer in the range 0 through 2147483647.

rand

returns a value in the range 0 through 1.0.

## random, irandm, drandm

Return the next random number value.

## Synopsis

```
real function random(flag)
integer flag
integer function irandm(flag)
integer flag
```



```
double precision function drandm(flag)
integer flag
```

## Description

If the argument, `flag`, is nonzero, the random number generator is restarted before the next random number is generated.

Integer values range from 0 through 2147483647.

Floating point values range from 0.0 through 1.0.

## range

Range functions.

## Synopsis

```
real function flmin()
real function flmax()
real function ffrac()
double precision function dflmin()
double precision function dflmax()
double precision function dffrac()
integer function inmax()
```

## Description

The following range functions return a value from a range of values.

**flmin**  
minimum single precision value

**flmax**  
maximum single precision value

**frac**  
smallest positive single precision value

**dflmin**  
minimum double precision value

**dflmax**  
maximum double precision value

**dfrac**  
smallest positive double precision value

**inmax**  
maximum integer

## rename

Rename a file.

## Synopsis

```
integer function rename(from, to)
character*(*) from
character*(*) to
```

## Description

Renames the existing file from where the new name is, the `from` value, to what you want it to be, the `to` value.. If the rename is successful, zero is returned; otherwise, the return value is an error code.

## rindex

Return index of substring.

## Synopsis

```
integer function rindex(a1, a2)
character*(*) a1
character*(*) a2
```

## Description

Return the index of the last occurrence of string `a2` in string `a1`.

## secnds, dsecnds

Return elapsed time.

## Synopsis

```
real function secnds(x)
real x
double precision function dsecnds(x)
double precision x
```

## Description

Returns the elapsed time, in seconds, since midnight, minus the value of `x`.

## setvbuf

Change I/O buffering behavior.

## Synopsis

```
interface
function setvbuf(lu, typ, size, buf)
integer setvbuf, lu, typ, size
character*(*) buf
end function
end interface
```

## Description

Fortran I/O supports 3 types of buffering:

- Fully buffered: on output, data is written once the buffer is full. On input, the buffer is filled when an input operation is requested and the buffer is empty.
- Line buffered: on output, data is written when a newline character is inserted in the buffer or when the buffer is full. On input, if an input operation is encountered and the buffer is empty, the buffer is filled until a newline character is encountered.
- Unbuffered: No buffer is used. Each I/O operation is completed as soon as possible. In this case, the `typ` and `size` arguments are ignored.

Logical units 5 (`stdin`) and 6 (`stdout`) are line buffered. Logical unit 0 (`stderr`) is unbuffered. Disk files are fully buffered. These defaults generally give the expected behavior. You can use `setvbuf3f` to change a unit's buffering type and size of the buffer.

### Note

---

The underlying `stdio` implementation may silently restrict your choice of buffer size.

This function must be called after the unit is opened and before any I/O is done on the unit.

The `typ` parameter can have the following values, 0 specifies full buffering, 1 specifies line buffering, and 2 specifies unbuffered. The `size` parameter specifies the size of the buffer. Note, the underlying `stdio` implementation may silently restrict your choice of buffer size.

The `buf` parameter is the address of the new buffer.

### Note

---

The buffer specified by the `buf` and `size` parameters must remain available to the Fortran runtime until after the logical unit is closed.

This function returns zero on success and non-zero on failure.

An example of a program in which this function might be useful is a long-running program that periodically writes a small amount of data to a log file. If the log file is line buffered, you could check the log file for progress. If the log file is fully buffered (the default), the data may not be written to disk until the program terminates.

## setvbuf3f

Change I/O buffering behavior.

## Synopsis

```
interface
function setvbuf3f(lu, typ, size)
integer setvbuf3f, lu, typ, size
```

```
end function
end interface
```

## Description

Fortran I/O supports 3 types of buffering., described in detail in the description of “[setvbuf,](#)” on page 256 Logical units 5 (stdin) and 6 (stdout) are line buffered. Logical unit 0 (stderr) is unbuffered. Disk files are fully buffered. These defaults generally give the expected behavior. You can use `setvbuf3f` to change a unit's buffering type and size of the buffer.

### Note

---

The underlying stdio implementation may silently restrict your choice of buffer size.

This function must be called after the unit is opened and before any I/O is done on the unit.

The `typ` parameter can have the following values, 0 specifies full buffering, 1 specifies line buffering, and 2 specifies unbuffered. The `size` parameter specifies the size of the buffer.

This function returns zero on success and non-zero on failure.

An example of a program in which this function might be useful is a long-running program that periodically writes a small amount of data to a log file. If the log file is line buffered, you could check the log file for progress. If the log file is fully buffered (the default), the data may not be written to disk until the program terminates.

## signal

Signal facility.

## Synopsis

```
integer function signal(signum, proc, flag)
integer signum
external proc
integer flag
```

## Description

`signal` allows the calling process to choose how the receipt of a specific signal is handled; `signum` is the signal and `proc` is the choice. If `flag` is negative, `proc` is a Fortran subprogram and is established as the signal handler for the signal. Otherwise, `proc` is ignored and the value of `flag` is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. There are two special cases of `flag`: 0 means use the default action and 1 means ignore this signal.

The return value is the previous action. If this is a value greater than one, then it is the address of a routine that was to have been called. The return value can be used in subsequent calls to `signal` to restore a previous action. A negative return value indicates a system error.

## sleep

Suspend execution for a period of time.

## Synopsis

```
subroutine sleep(itime)
integer itime
```

## Description

Suspends the process for t seconds.

## stat, lstat, fstat, fstat64

Get file status.

## Synopsis

```
integer function stat(nm, statb)
character*(*) nm
integer statb(*)

integer function lstat(nm, statb)
character*(*) nm
integer statb(*)

integer function fstat(lu, statb)
integer lu
integer statb(*)

integer function fstat64(lu, statb)
integer lu
integer*8 statb(*)
```

## Description

Return the file status of the file in the array statb. If successful, zero is returned; otherwise, the value of -1 is returned. stat obtains information about the file whose name is nm; if the file is a symbolic link, information is obtained about the file the link references. lstat is similar to stat except lstat returns information about the link. fstat obtains information about the file which is connected to logical unit lu.

## stime

Set time.

## Synopsis

```
integer function stime(tp)
integer tp
```

## Description

Set the system time and date. tp is the value of the time measured in seconds from 00:00:00 GMT January 1, 1970.

## symlink

Make symbolic link.

## Synopsis

```
integer function symlink(n1, n2)
character* (*) n1
character* (*) n2
```

## Description

Create a symbolic link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

## system

Issue a shell command.

## Synopsis

```
integer function system(str)
character* (*) str
```

## Description

system causes the string, str, to be given to the shell as input. The current process waits until the shell has completed and returns the exit status of the shell.

## time

Return system time.

## Synopsis

```
integer*8 function time()
```

## Description

Return the time since 00:00:00 GMT, January 1, 1970, measured in seconds.

## times

Get process and child process time

## Synopsis

```
integer function times(buff)
integer buff (*)
```

## Description

Returns the time-accounting information for the current process and for any terminated child processes of the current process in the array buff. If successful, zero is returned; otherwise, the negation of the error code is returned.

## ttynam

Get name of a terminal

### Synopsis

```
character*(*) ttynam(lu)
integer lu
```

### Description

Returns a blank padded path name of the terminal device connected to the logical unit lu. The lu is not connected to a terminal, blanks are returned.

## unlink

Remove a file.

### Synopsis

```
integer function unlink(fil)
character*(*) fil
```

### Description

Removes the file specified by the pathname fil. If successful, zero is returned; otherwise, an error code is returned.

## wait

Wait for process to terminate.

### Synopsis

```
integer function wait(st)
integer st
```

### Description

wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last wait, return is immediate. If there are no child processes, return is immediate with an error code.

If the return value is positive, it is the process id of the child and st is its termination status. If the return value is negative, it is the negation of an error code.

## VAX System Subroutines

The PGI FORTRAN77 compiler, pgf77, supports a variety of built-in functions and VAX/VMS system subroutines.

## Built-In Functions

The built-in functions perform inter-language utilities for argument passing and location calculations. The following built-in functions are available:

### %LOC(arg)

Compute the address of the argument arg.

### %REF(a)

Pass the argument a by reference.

### %VAL(a)

Pass the argument as a 32-bit immediate value (64-bit if a is double precision.) A value of 64-bits is also possible if supported for integer and logical values.

## VAX/VMS System Subroutines

### DATE

The DATE subroutine returns a nine-byte string containing the ASCII representation of the current date. It has the form:

```
CALL DATE(buf)
```

where buf is a nine-byte variable, array, array element, or character substring. The date is returned as a nine-byte ASCII character string of the form:

```
dd-mmm-yy
```

Where:

dd

is the two-digit day of the month

mmm

is the three-character abbreviation of the month

yy

is the last two digits of the year

### EXIT

The EXIT subroutine causes program termination, closes all open files, and returns control to the operating system. It has the form:

```
CALL EXIT[(exit_status)]
```

where:

exit\_status

is an optional integer argument used to specify the image exit value



## GETARG

The GETARG subroutine returns the Nth command line argument in character variable ARG. For N equal to zero, the name of the program is returned.

```
SUBROUTINE GETARG(N, ARG)
INTEGER*4 N
CHARACTER*(*) ARG
```

## IARGC

The IARGC subroutine returns the number of command line arguments following the program name.

```
INTEGER*4 FUNCTION IARGC()
```

## IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. It has the form:

```
CALL IDATE(IMONTH, IDAY, IYEAR)
```

If the current date were October 9, 2004, the values of the integer variables upon return would be:

```
IMONTH = 10
IDAY = 9
IYEAR = 04
```

## MVBITS

The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). MVBITS transfers  $a_3$  bits from positions  $a_2$  through  $(a_2 + a_3 - 1)$  of the source, *src*, to positions  $a_5$  through  $(a_5 + a_3 - 1)$  of the destination, *dest*. Other bits of the destination location remain unchanged. The values of  $(a_2 + a_3)$  and  $(a_5 + a_3)$  must be less than or equal to 32 (less than or equal to 64 if the source or destination is INTEGER\*8). It has the form:

```
CALL MVBITS(src, a2, a3, dest, a5)
```

Where:

*src*

is an integer variable or array element that represents the source location.

$a_2$

is an integer expression that identifies the first position in the field transferred from *src*.

$a_3$

is an integer expression that identifies the length of the field transferred from *src*.

*dest*

is an integer variable or array element that represents the destination location.

$a_5$

is an integer expression that identifies the starting position within *dest*, for the bits being transferred.

## RAN

The RAN subroutine returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1. The result is a floating point number that is uniformly distributed in the range between 0.0 and 1.0 exclusive. It has the form:

```
y = RAN(i)
```

where y is set equal to the value associated by the function with the seed argument i. The argument i must be an INTEGER\*4 variable or INTEGER\*4 array element.

The argument i should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 6969 * SEED + 1 ! MOD
2**32
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

If the command-line option to treat all REAL declarations as DOUBLE PRECISION declarations is in effect, RAN returns a DOUBLE PRECISION value.

## SECNDS

The SECNDS subroutine provides system time of day, or elapsed time, as a floating point value in seconds. It has the form:

```
y = SECNDS(x)
```

where (REAL or DOUBLE PRECISION) y is set equal to the time in seconds since midnight, minus the user supplied value of the (REAL or DOUBLE PRECISION) x. Elapsed time computations can be performed with the following sequence of calls.

```
X = SECNDS(0.0)
...
... ! Code to be timed
...
DELTA = SECNDS(X)
```

The accuracy of this call is the same as the resolution of the system clock.

## TIME

The TIME subroutine returns the current system time as an ASCII string. It has the form:

```
CALL TIME(buf)
```

where buf is an eight-byte variable, array, array element, or character substring. The TIME call returns the time as an eight-byte ASCII character string of the form:

```
hh:mm:ss
```

For example:

16:45:23

---

Note that a 24-hour clock is used.



# Chapter 8. OpenMP Directives for Fortran

The PGF77 and PGF95 Fortran compilers support the OpenMP Fortran Application Program Interface. The OpenMP shared-memory parallel programming model is defined by a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran programs.

The directives include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs. The data environment is controlled using clauses on the directives or with additional directives. Run-time library routines are provided to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region. Finally, environment variables are provided to control the execution behavior of parallel programs. For more information on OpenMP, see

<http://www.openmp.org>

For an introduction to how to execute programs that use multiple processors along with some pointers to example code, see “Parallel Programming Using PGI Compilers” in the PGI User’s Guide.

## OpenMP Overview

Let’s look at the OpenMP shared-memory parallel programming model and some common OpenMP terminology.

### OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model is a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran, C and C++ programs.

#### Fortran directives

Allow users to place hints in the source code to help the compiler generate more efficient code. You typically use directives to control the actions of the compiler in a particular portion of a program without

affecting the program as a whole. You place them in your source code where you want them to take effect; and they usually stay in effect from the point where included until the end of the compilation unit or until another directive or pragma changes its status.

Fortran directives and C/C++ pragmas include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs.

## Note

---

The data environment is controlled either by using clauses on the directives or with additional directives.

### Run-time library routines

Are available to query the parallel run-time environment, for example to determine how many threads are participating in execution of a parallel region.

### Environment variables

Are available to control the execution behavior of parallel programs. For more information on OpenMP, see [www.openmp.org](http://www.openmp.org).

## Terminology

For OpenMP 3.0 there are a number of terms for which it is useful to have common definitions.

### Thread

An execution entity with a stack and associated static memory, called *threadprivate memory*.

- An OpenMP thread is a thread that is managed by the OpenMP runtime system.
- A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

### Region

All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. PGI currently does not support nested parallel regions.

### Parallel region

In OpenMP 3.0 there is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

- An inactive parallel region is executed by a single thread.
- An active parallel region is a parallel region that is executed by a team consisting of more than one thread.

## Note

The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.0. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

```

program test
  logical omp_in_parallel

!$omp parallel
  print *, omp_in_parallel()
!$omp end parallel

  stop
end

```

Suppose we run this program with OMP\_NUM\_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.0, the program yields F. In OpenMP 3.0, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

PGI currently does not support nested parallel regions so currently has only one level of active parallel regions.

### Task

A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

## OpenMP Example

Look at the following simple OpenMP example involving loops.

### Example 8.1. OpenMP Loop Example

```

PROGRAM MAIN
  INTEGER I, N, OMP_GET_THREAD_NUM
  REAL*8 V(1000), GSUM, LSUM

  GSUM = 0.0D0
  N = 1000

  DO I = 1, N
    V(I) = DBLE(I)
  ENDDO

!$OMP PARALLEL PRIVATE(I,LSUM) SHARED(V,GSUM,N)
  LSUM = 0.0D0
!$OMP DO
  DO I = 1, N
    LSUM = LSUM + V(I)
  ENDDO
!$OMP END DO
!$OMP CRITICAL
  print *, "Thread ",OMP_GET_THREAD_NUM()," local sum: ",LSUM
  GSUM = GSUM + LSUM
!$OMP END CRITICAL

```

```
!$OMP END PARALLEL

PRINT *, "Global Sum: ", GSUM

STOP
END
```

If you execute this example with the environment variable `OMP_NUM_THREADS` set to 4, then the output looks similar to this:

```
Thread      0 local sum:    31375.00000000000
Thread      1 local sum:    93875.00000000000
Thread      2 local sum:   156375.00000000000
Thread      3 local sum:   218875.00000000000
Global Sum: 500500.00000000000
FORTRAN STOP
```

## Task Overview

Every part of an OpenMP program is part of a task. A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- **Packaging:** Each encountering thread packages a new instance of a task - code and data.
- **Execution:** Some thread in the team executes the task at some later time.

In the following sections, we use this terminology:

### Task

The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

- An explicit task is a task generated when a task construct is encountered during execution.
- An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

### Task construct

A task directive plus a structured block

### Task region

The dynamic sequence of instructions produced by the execution of a task by a thread.

## Tasks

Every part of an OpenMP program is part of a task. [“Task Overview,” on page 270](#) provides a general overview of tasks and general terminology associated with tasks. This section provides more detailed information about tasks, including tasks scheduling points and the task construct.



## Task Characteristics and Activities

A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

## Task Scheduling Points

PGI currently supports four task scheduling points: at the beginning of a task, at the end of a task, a taskwait, and at a barrier.

- Beginning of a task.

At the beginning of a task, the task can be executed immediately or registered for later execution. A programmer-specified "if" clause that is FALSE forces immediate execution of the task. The implementation can also force immediate execution; for example, a task within a task is never registered for later execution, it executes immediately.

- End of a task

At the end of a task, the behavior of the scheduling point depends on how the task was executed. If the task was immediately executed, execution continues to the next statement. If it was previously registered and is being executed "out of sequence", control returns to where the task was executed - a taskwait.

- Taskwait

A taskwait executes all registered tasks at the time it is called. In addition to executing all tasks registered by the calling thread, it also executes tasks previously registered by other threads. Let's take a quick look at this process.

Suppose thread 0 called taskwait and is executing tasks and that thread 1 is registering tasks. Depending on the timing between thread 0 and thread 1, thread 0 may execute none of the tasks, all of the tasks, or some of tasks.

### Note

---

Taskwait waits only for immediate children tasks, not for descendant tasks. You can achieve waiting on descendants but ensuring that each child also waits on its children.

- Barrier

A barrier can be explicit or implicit. An example of an implicit barrier is the end of a parallel region.

The barrier effectively contains taskwaits. All threads must arrive at the barrier for the barrier to complete. This rule guarantees that all tasks have been executed at the completion of the barrier.

## Task Construct

A task construct is a task directive plus a structured block, with the following syntax:

```
#pragma omp task [clause[[],]clause] ...]
    structured-block
```

where clause can be one of the following:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
```

Consider the following simple example of a program using tasks. This example illustrates the difference between registering tasks and executing tasks, a concept that is fundamental to understanding tasks.

This program contains a parallel region that contains a single region. The single region contains a loop that registers 10 tasks. Before reading the explanation that follows the example, consider what happens if you use four threads with this example.

### Example 8.2. OpenMP Task Fortran Example

```
PROGRAM MAIN
  INTEGER I
  INTEGER omp_get_thread_num
!$OMP PARALLEL PRIVATE(I)
!$OMP SINGLE
  DO I = 1, 10
    CALL SLEEP(MOD(I,2))
    PRINT *, "TASK ", I, " REGISTERED BY THREAD ", omp_get_thread_num()
!$OMP TASK FIRSTPRIVATE(I)
    CALL SLEEP(MOD(I,5))
    PRINT *, "TASK ", I, " EXECUTED BY THREAD ", omp_get_thread_num()
!$OMP END TASK
  ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
END
```

If you run this program with four threads, 0 through 3, one thread is in the single region registering tasks. The other three threads are in the implied barrier at the end of the single region executing tasks. Further, when the thread executing the single region completes registering the tasks, it joins the other threads and executes tasks.

The program includes calls to `sleep` to slow the program and allow all threads to participate.

The output for the Fortran example is similar to the following. In this output, thread 1 was registering tasks while the other three threads - 0, 2, and 3 - were executing tasks. When all 10 tasks were registered, thread 1 began executing tasks as well.

```

TASK 1 REGISTERED BY THREAD 1
TASK 2 REGISTERED BY THREAD 1
TASK 1 EXECUTED BY THREAD 0
TASK 3 REGISTERED BY THREAD 1
TASK 4 REGISTERED BY THREAD 1
TASK 2 EXECUTED BY THREAD 3
TASK 5 REGISTERED BY THREAD 1
TASK 6 REGISTERED BY THREAD 1
TASK 6 EXECUTED BY THREAD 3
TASK 5 EXECUTED BY THREAD 3
TASK 7 REGISTERED BY THREAD 1
TASK 8 REGISTERED BY THREAD 1
TASK 3 EXECUTED BY THREAD 0
TASK 9 REGISTERED BY THREAD 1
TASK 10 REGISTERED BY THREAD 1
TASK 10 EXECUTED BY THREAD 1
TASK 4 EXECUTED BY THREAD 2
TASK 7 EXECUTED BY THREAD 0
TASK 8 EXECUTED BY THREAD 3
TASK 9 EXECUTED BY THREAD 1

```

## Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the PGI Fortran compilers when the option `-mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

With the exception of the SGI-compatible DOACROSS directive, the *sentinel* must comply with these rules:

- Be one of these: `!$OMP`, `C$OMP`, or `*$OMP`.
- Must start in column 1 (one).
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is `C$`.

In addition to the sentinel rules, the directive must also comply with these rules:

- Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- Initial directive lines must have a space or zero in column six.
- Continuation directive lines must have a character other than a space or a zero in column six. Continuation lines for `C$DOACROSS` directives are specified using the `C$&` sentinel.
- Directives which are presented in pairs must be used in pairs.

Valid clauses depend on the directive. Clauses associated with directives have these characteristics:

- The order in which clauses appear in the parallelization directives is not significant.
- Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.

- Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

## Directive Recognition

The compiler option `-mp` enables recognition of the parallelization directives. The use of this option also implies:

`-Mreentrant`

Local variables are placed on the stack and optimizations, such as `-Mnoframe`, that may result in non-reentrant code are disabled.

`-Miomutex`

Critical sections are generated around Fortran I/O statements.

Many of the directives are presented in pairs and must be used in pairs. In the examples given with each section, the routines `omp_get_num_threads()` and `omp_get_thread_num()` are used; refer to [Runtime Library Routines](#) for more information. These routines return the number of threads currently in the team executing the parallel region and the thread number within the team, respectively.

## Directive Clauses

Some directives accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives, so the clauses that are valid are included with the description of the directive. Typically, if no data scope clause is specified for variables, the default scope is *share*.

The following table provides a brief summary of the clauses associated with OpenMP directives that PGI supports. Following the table is more detailed information about these clauses. For complete information on OpenMP and use of these clauses, refer to the PGI User's Guide and to the OpenMP documentation available on the WorldWide Web.

Table 8.1. Directive Clauses Summary Table

Clause	Applies to	Description
<a href="#">COLLAPSE (n)</a>	DO...END DO PARALLEL DO ... END PARALLEL DO PARALLEL WORKSHARE	Specifies how many loops are associated with the loop construct.
<a href="#">COPYIN (list)</a>	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region. Then, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

Clause	Applies to	Description
<code>COPYPRIVATE(list)</code>	END SINGLE	Specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.
<code>DEFAULT</code>	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.
<code>FIRSTPRIVATE(list)</code>	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	Specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.
<code>IF()</code>	PARALLEL ... END PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies whether a loop should be executed in parallel or in serial.
<code>LASTPRIVATE(list)</code>	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	Specifies that the enclosing context's version of the variable is set equal to the <i>private</i> version of whichever thread executes the final iteration of a for-loop construct.
<code>NOWAIT</code>	DO ... END DO SECTIONS SINGLE WORKSHARE ... END WORKSHARE	Overrides the barrier implicit in a directive.

Clause	Applies to	Description
<b>NUM_THREADS</b>	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Sets the number of threads in a thread team.
<b>ORDERED</b>	DO...END DO PARALLEL DO ... END PARALLEL DO	Required on a parallel FOR statement if an ordered directive is used in the loop.
<b>PRIVATE</b>	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	Specifies that each thread should have its own instance of a variable.
<b>REDUCTION</b> ({operator   intrinsic } : list)	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
<b>SCHEDULE</b> (type [,chunk])	DO ... END DO PARALLEL DO... END PARALLEL DO	Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.
<b>SHARED</b>	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies that one or more variables should be shared among all threads. All threads within a team access the same storage area for shared variables
<b>UNTIED</b>	TASK TASKWAIT	Specifies that any thread in the team can resume the task region after a suspension.

## COLLAPSE (n)

The COLLAPSE(n) clause specifies how many loops are associated with the loop construct.

The parameter of the collapse clause must be a constant positive integer expression. If no COLLAPSE clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space, which is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If the loop directive contains a COLLAPSE clause then there may be more than one associated loop.

## COPYIN (list)

The COPYIN(list) clause allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region; that is, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

The COPYIN clause applies only to THREADPRIVATE common blocks. If you specify a COPYIN clause, here are a few tips:

- You cannot specify the same entity name more than once in the list.
- You cannot specify the same entity name in separate COPYIN clauses of the same directive.
- You cannot specify both a common block name and any variable within that same named common block in the list.
- You cannot specify both a common block name and any variable within that same named common block in separate COPYIN clauses of the same directive.

## COPYPRIVATE(list)

The COPYPRIVATE(list) clause specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.

You use a COPYPRIVATE(list) clause on an END SINGLE directive to cause the variables in the list to be copied from the private copies in the single thread that executes the SINGLE region to the other copies in all other threads of the team at the end of the SINGLE region.

### Note

---

The COPYPRIVATE clause must not appear on the same END SINGLE directive as a NOWAIT clause.

The compiler evaluates a COPYPRIVATE clause before any threads have passed the implied BARRIER directive at the end of that construct.

## DEFAULT

The `DEFAULT` clause specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables. The `DEFAULT` clause lets you specify the default attribute for variables in the lexical extent of the parallel region. Individual clauses specifying `PRIVATE`, `SHARED`, and so on, override the declared `DEFAULT`.

Specifying `DEFAULT(NONE)` declares that there is no implicit default. With this declaration, each variable in the parallel region must be explicitly listed with an attribute of `PRIVATE`, `SHARED`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION`.

## FIRSTPRIVATE(list)

The `FIRSTPRIVATE(list)` clause specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.

Variables that appear in the list of a `FIRSTPRIVATE` clause are subject to the same semantics as `PRIVATE` variables; however, these variables are initialized from the original object that exists prior to entering the parallel region.

If a directive construct contains a `FIRSTPRIVATE` argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

## IF()

The `IF()` clause specifies whether a loop should be executed in parallel or in serial.

In the presence of an `IF` clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to `.TRUE.`. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

## LASTPRIVATE(list)

The `LASTPRIVATE(list)` clause specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration (for-loop construct).

## NOWAIT

The `NOWAIT` clause overrides the barrier implicit in a directive. When you specify `NOWAIT`, it removes the implicit barrier synchronization at the end of a for or sections construct.

## NUM\_THREADS

The `NUM_THREADS` clause sets the number of threads in a thread team. The `num_threads` clause allows a user to request a specific number of threads for a parallel construct. If the `num_threads` clause is present, then



## ORDERED

The **ORDERED** clause specifies that a loop is executed in the order of the loop iterations. This clause is required on a parallel FOR statement when an ordered directive is used in the loop.

You use this clause 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.

## PRIVATE

The **PRIVATE** clause specifies that each thread should have its own instance of a variable. Therefore, variables specified in a **PRIVATE** list are private to each thread in a team. In effect, the compiler creates a separate copy of each of these variables for each thread in the team. When an assignment to a private variable occurs, each thread assigns to its local copy of the variable. When operations involving a private variable occur, each thread performs the operations using its local copy of the variable.

Tips about private variables:

- Variables declared private in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand-side expression, the results of the expression will be undefined, indicating the probability of a coding error.
- Variables declared private in a parallel region are undefined when serial execution resumes at the end of the parallel region.

## REDUCTION

The **REDUCTION** clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. updates named variables declared on the clause within the directive construct.

Intermediate values of **REDUCTION** variables are not used within the parallel construct, other than in the updates themselves. Variables that appear in the list of a **REDUCTION** clause must be **SHARED**. A private copy of each variable in `list` is created for each thread as if the **PRIVATE** clause had been specified. Each private copy is initialized according to the operator as specified in the following table:

Table 8.2. Initialization of REDUCTION Variables

Operator / Intrinsic	Initialization	Operator / Intrinsic	Initialization
+	0	.NEQV.	.FALSE.
*	1	MAX	Smallest representable number
-	0	MIN	Largest representable number
.AND.	.TRUE.	IAND	All bits on

Operator / Intrinsic	Initialization		Operator / Intrinsic	Initialization
.OR.	.FALSE.		IOR	0
.EQV.	.TRUE.		IEOR	0

At the end of the parallel region, a reduction is performed on the instances of variables appearing in `list` using operator or intrinsic as specified in the REDUCTION clause. The initial value of each REDUCTION variable is included in the reduction operation. If the `{operator | intrinsic}`: portion of the REDUCTION clause is omitted, the default reduction operator is "+" (addition).

## SCHEDULE

The SCHEDULE clause specifies how iterations of the DO loop are divided up between processors. Given a SCHEDULE (type [, chunk]) clause, the type can be STATIC, DYNAMIC, GUIDED, or RUNTIME, defined in the following list.

- When SCHEDULE (STATIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. The blocks of iterations are statically assigned to threads in a round-robin fashion in order of the thread ID numbers. The chunk must be a scalar integer expression. If chunk is not specified, a default chunk size is chosen equal to:

```
(number_of_iterations + omp_num_threads() - 1) / omp_num_threads()
```

- When SCHEDULE (DYNAMIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. The chunk must be a scalar integer expression. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (GUIDED, chunk) is specified, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. Chunk specifies the minimum number of iterations to dispatch each time, except when there are less than chunk iterations remaining to be processed, at which point all remaining iterations are assigned. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (RUNTIME) is specified, the decision regarding iteration scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the OMP\_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is equivalent to SCHEDULE(STATIC).

## SHARED

The SHARED clause specifies variables that must be available to all threads. If you specify a variable as SHARED, you are stating that all threads can safely share a single copy of the variable. When one or more variables are shared among all threads, all threads access the same storage area for the shared variables.

## UNTIED

The UNTIED clause specifies that any thread in the team can resume the task region after a suspension.

## Note

The thread number may change at any time during the execution of an untied task. Therefore, the value returned by `omp_get_thread_num` is generally not useful during execution of such a task region.

## Directive Summary Table

Table 8.3 provides a brief summary of the directives and pragmas that PGI supports.

Table 8.3. Directive Summary Table

Directive	Description
<code>ATOMIC</code>	Semantically equivalent to enclosing a single statement in the <code>CRITICAL...END CRITICAL</code> directive. Note: Only certain statements are allowed.
<code>BARRIER</code>	Synchronizes all threads at a specific point in a program so that all threads complete work to that point before any thread continues.
<code>CRITICAL ... END CRITICAL</code>	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
<code>C\$DOACROSS</code>	Provides a mechanism for distribution of loop iterations across the available threads in a parallel region.
<code>DO...END DO</code>	Specifies that the compiler should parallelize the loop to which it applies, even though that loop is not contained within a parallel region.
<code>FLUSH</code>	When this appears, all processor-visible data items, or, when a list is present ( <code>FLUSH [list]</code> ), only those specified in the list, are written to memory, thus ensuring that all the threads in a team have a consistent view of certain objects in memory.
<code>MASTER ... END MASTER</code>	Designates code that executes on the master thread and that is skipped by the other threads.
<code>ORDERED</code>	Defines a code block that is executed by only one thread at a time, and in the order of the loop iterations; this makes the ordered code block sequential, while allowing parallel execution of statements outside the code block.
<code>PARALLEL ... END PARALLEL</code>	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
<code>PARALLEL DO</code>	Enables you to specify which loops the compiler should parallelize.
<code>PARALLEL SECTIONS</code>	Defines a non-iterative work-sharing construct without the need to define an enclosing parallel region.
<code>PARALLEL WORKSHARE</code>	Provides a short form method for including a <code>WORKSHARE</code> directive inside a <code>PARALLEL</code> construct.

Directive	Description
<code>SECTIONS ... END SECTIONS</code>	Defines a non-iterative work-sharing construct within a parallel region.
<code>SINGLE ... END SINGLE</code>	Designates code that executes on a single thread and that is skipped by the other threads.
<code>TASK</code>	Defines an explicit task.
<code>TASKWAIT</code>	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
<code>THREADPRIVATE</code>	When a common block or variable that is initialized appears in this directive, each thread's copy is initialized once prior to its first use.
<code>WORKSHARE ... END WORKSHARE</code>	Provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

## ATOMIC

The OpenMP `ATOMIC` directive is semantically equivalent to a single statement in a `CRITICAL...END CRITICAL` directive.

### Syntax:

```
!$OMP ATOMIC
```

### Usage:

The `ATOMIC` directive is semantically equivalent to enclosing the following single statement in a `CRITICAL / END CRITICAL` directive pair.

The statements must be one of the following forms:

```
x = x operator expr
```

```
x = intrinsic (x, expr)
```

```
x = expr operator x
```

```
x = intrinsic (expr, x)
```

where `x` is a scalar variable of intrinsic type, `expr` is a scalar expression that does not reference `x`, `intrinsic` is one of `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`, and `operator` is one of `+`, `*`, `-`, `/`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`

## BARRIER

The OpenMP `BARRIER` directive defines a point in a program where each thread waits for all other threads to arrive before continuing with program execution.

### Syntax:

```
!$OMP BARRIER
```

### Usage:

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The `BARRIER` directive synchronizes all threads at such a point in a

program. Multiple barrier points are allowed within a parallel region. The BARRIER directive must either be executed by all threads executing the parallel region or by none of them.

## CRITICAL ... END CRITICAL

The CRITICAL...END CRITICAL directive requires a thread to wait until no other thread is executing within a critical section.

### Syntax:

```
!$OMP CRITICAL [(name)]
  < Fortran code executed in body of critical section >
!$OMP END CRITICAL [(name)]
```

### Usage:

Within a parallel region, there may exist subregions of code that will not execute properly when executed by multiple threads simultaneously. This issue is often due to a shared variable that is written and then read again.

The CRITICAL... END CRITICAL directive pair defines a subsection of code within a parallel region, referred to as a critical section, which is executed one thread at a time.

The first thread to arrive at a critical section is the first to execute the code within the section. The second thread to arrive does not begin execution of statements in the critical section until the first thread exits the critical section. Likewise, each of the remaining threads wait its turn to execute the statements in the critical section.

You can use the optional name argument to identify the critical region. Names that identify critical regions have external linkage and are in a name space separate from the name spaces used by labels, tags, members, and ordinary identifiers. If a name argument appears on a CRITICAL directive, the same name must appear on the END CRITICAL directive.

### Note

Critical sections cannot be nested, and any such specifications are ignored. Branching into or out of a critical section is illegal.

### Example of Critical...End Critical directive:

```
PROGRAM CRITICAL_USE
  REAL A(100,100),MX, LMX
  INTEGER I, J
  MX = -1.0
  LMX = -1.0
  CALL RANDOM_SEED( )
  CALL RANDOM_NUMBER(A)
!$OMP PARALLEL PRIVATE(I), FIRSTPRIVATE(LMX)
!$OMP DO
  DO J=1,100
    DO I=1,100
      LMX = MAX(A(I,J),LMX)
    ENDDO
  ENDDO
!$OMP CRITICAL
  MX = MAX(MX,LMX)
!$OMP END CRITICAL
```

```
!$OMP END PARALLEL
  PRINT *, "MAX VALUE OF A IS ", MX
END
```

This program could also be implemented without the critical region by declaring `MX` as a reduction variable and performing the `MAX` calculation in the loop using `MX` directly rather than using `LMX`. Refer to “[PARALLEL ... END PARALLEL](#)” and “[DO...END DO](#)” for more information on how to use the `REDUCTION` clause on a parallel `DO` loop.

## C\$DOACROSS

The `C$DOACROSS` directive, while not part of the OpenMP standard, is supported for compatibility with programs parallelized using legacy SGI-style directives.

### Syntax:

```
C$DOACROSS [ Clauses ]
  < Fortran DO loop to be executed in parallel >
```

### Clauses:

<code>{PRIVATE   LOCAL} (list)</code>	<code>CHUNK=&lt;integer_expression&gt;</code>
<code>{SHARED   SHARE} (list)</code>	<code>IF (logical_expression)</code>
<code>MP_SCHEDTYPE={SIMPLE   INTERLEAVE}</code>	

### Usage:

The `C$DOACROSS` directive has the effect of a combined parallel region and parallel `DO` loop applied to the loop immediately following the directive. It is very similar to the OpenMP `PARALLEL DO` directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort.

The `C$DOACROSS` directive must not appear within a parallel region. It is a shorthand notation that tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region.

### Important

---

While The `C$DOACROSS` syntax may be more convenient, if multiple successive `DO` loops are to be parallelized, it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP `DO` directive.

A variable declared `PRIVATE` or `LOCAL` to a `C$DOACROSS` loop is treated the same as a private variable in a parallel region or `DO`. A variable declared `SHARED` or `SHARE` to a `C$DOACROSS` loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as `PRIVATE` in a parallel region or `DO`. This same default status is used in `C$DOACROSS` loops as well.

For more information on clauses, refer to “[Directive Clauses,](#)” on page 274.

## DO...END DO

The OpenMP `DO...END DO` directive support parallel execution and the distribution of loop iterations across available threads in a parallel region.

**Syntax:**

```
!$OMP DO [Clauses]
  < Fortran DO loop to be executed in parallel >
!$OMP END DO [NOWAIT]
```

**Clauses:**

PRIVATE(list)	SCHEDULE (type [, chunk])
FIRSTPRIVATE(list)	COLLAPSE (n)
LASTPRIVATE(list)	ORDERED
REDUCTION({operator   intrinsic} : list)	

**Usage:**

The real purpose of supporting parallel execution is the distribution of work across the available threads. The DO... END DO directive pair provides a convenient mechanism for the distribution of loop iterations across the available threads in a parallel region.

While you can explicitly manage work distribution with constructs such as the following one, these constructs are not in the form of directives.

**Examples:**

```
IF (omp_get_thread_num() .EQ. 0)
THEN
  ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
  ...
ENDIF
```

**Tips**

Remember these items about clauses in the DO...END DO directives:

- Variables declared in a PRIVATE list are treated as private to each thread participating in parallel execution of the loop, meaning that a separate copy of the variable exists with each thread.
- Variables declared in a FIRSTPRIVATE list are PRIVATE, and are initialized from the original object existing before the construct.
- Variables declared in a LASTPRIVATE list are PRIVATE, and the thread that executes the sequentially last iteration updates the version of the object that existed before the construct.
- The REDUCTION clause for the directive is described in [“Directive Clauses,” on page 274](#).
- The SCHEDULE clause specifies how iterations of the DO loop are divided up between threads. For more information on this clause, refer to [“Directive Clauses,” on page 274](#).
- If ORDERED code blocks are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ORDERED code blocks, refer to [“ORDERED”](#).
- The DO... END DO directive pair directs the compiler to distribute the iterative DO loop immediately following the !\$OMP DO directive across the threads available to the program. The DO loop is executed in parallel by the team that was started by an enclosing parallel region. If the !\$OMP END DO directive is not

specified, the !\$OMP DO is assumed to end with the enclosed DO loop. DO... END DO directive pairs may not be nested. Branching into or out of a !\$OMP DO loop is not supported.

- By default, there is an implicit barrier after the end of the parallel loop; the first thread to complete its portion of the work waits until the other threads have finished their portion of work. If NOWAIT is specified, the threads will not synchronize at the end of the parallel loop.

In addition to the preceding items, remember these items about !\$OMP DO loops :

- The DO loop index variable is always private.
- !\$OMP DO loops must be executed by all threads participating in the parallel region or none at all.
- The END DO directive is optional, but if it is present it must appear immediately after the end of the enclosed DO loop.
- Values of the loop control expressions and the chunk expressions must be the same for all threads executing the loop.

### Example:

```
PROGRAM DO_USE
  REAL A(1000), B(1000)
  DO I=1,1000
    B(I) = FLOAT(I)
  ENDDO
 !$OMP PARALLEL
 !$OMP DO
  DO I=1,1000
    A(I) = SQRT(B(I));
  ENDDO
  ...
 !$OMP END PARALLEL
  ...
END
```

## FLUSH

The OpenMP FLUSH directive ensures that processor-visible data items are written back to memory at the point at which the directive appears.

### Syntax:

```
!$OMP FLUSH [(list)]
```

### Usage:

The OpenMP FLUSH directive ensures that all processor-visible data items, or only those specified in *list*, when it is present, are written back to memory at the point at which the directive appears.

## MASTER ... END MASTER

The MASTER...END MASTER directive allows the user to designate code that must execute on a master thread and that is skipped by other threads in the team of threads.

### Syntax:



```
!$OMP MASTER
  < Fortran code executed in body of MASTER section >
!$OMP END MASTER
```

**Usage:**

A master thread is a single thread of control that begins an OpenMP program and which is present for the duration of the program. In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the MASTER... END MASTER directive pair allows the user to conveniently designate code that executes on the master thread and is skipped by the other threads.

- There is no implied barrier on entry to or exit from a master section of code.
- Nested master sections are ignored.
- Branching into or out of a master section is not supported.

**Examples:**

Example of Fortran **MASTER...END MASTER** directive

```
PROGRAM MASTER_USE
  INTEGER A(0:1)
  INTEGER omp_get_thread_num
  A=-1
!$OMP PARALLEL
  A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP MASTER
  PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END MASTER
!$OMP END PARALLEL
  PRINT *, "A(0)=", A(0), " A(1)=", A(1)
END
```

**ORDERED**

The OpenMP ORDERED directive allows the user to identify a portion of code within an ordered code block that must be executed in the original, sequential order, while allowing parallel execution of statements outside the code block.

**Syntax:**

```
!$OMP ORDERED
  < Fortran code block executed by processor >
!$OMP END ORDERED
```

**Usage:**

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The structured code block between the ORDERED / END ORDERED directives is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ORDERED directive:

- The ordered code block must be a structured block.

- It is illegal to branch into or out of the block.
- A given iteration of a loop with a DO directive cannot execute the same ORDERED directive more than once, and cannot execute more than one ORDERED directive.

## PARALLEL ... END PARALLEL

The OpenMP PARALLEL...END PARALLEL directive supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.

### Syntax:

```
!$OMP PARALLEL [Clauses]
  < Fortran code executed in body of parallel region >
!$OMP END PARALLEL
```

### Clauses:

PRIVATE(list)	REDUCTION([operator   intrinsic]:) list
SHARED(list)	COPYIN(list)
DEFAULT(PRIVATE   SHARED   NONE)	IF(scalar_logical_expression)
FIRSTPRIVATE(list)	NUM_THREADS(scalar_integer_expression)

### Usage:

This directive pair declares a region of parallel execution. It directs the compiler to create an executable in which the statements within the structured block, such as between PARALLEL and PARALLEL END for directives, are executed by multiple lightweight threads. The code that lies within this structured block is called a *parallel region*.

The OpenMP parallelization directives support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered. At the entrance to the parallel region, a system-dependent number of symmetric parallel threads begin executing all statements in the parallel region redundantly. These threads share work by means of work-sharing constructs such as parallel DO loops or FOR loops.

- The number of threads in the team is controlled by the OMP\_NUM\_THREADS environment variable. If OMP\_NUM\_THREADS is not defined, the program executes parallel regions using only one processor.
- Branching into or out of a parallel region is not supported.
- All other shared-memory parallelization directives must occur within the scope of a parallel region. Nested PARALLEL... END PARALLEL directive pairs are not supported and are ignored.
- There is an implicit barrier at the end of the parallel region, which, in the directive, is denoted by the END PARALLEL directive. When all threads have completed execution of the parallel region, a single thread resumes execution of the statements that follow.

### Note

---

By default, there is no work distribution in a parallel region. Each active thread executes the entire region redundantly until it encounters a directive that specifies work distribution. For work distribution, see the “DO...END DO”, “PARALLEL DO”, or “C\$DOACROSS” directives.

## Example

### Example of Fortran **PARALLEL...END PARALLEL** directive

```

PROGRAM WHICH_PROCESSOR_AM_I
  INTEGER A(0:1)
  INTEGER omp_get_thread_num
  A(0) = -1
  A(1) = -1
!$OMP PARALLEL
  A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP END PARALLEL
  PRINT *, "A(0)=",A(0)," A(1)=",A(1)
END

```

### Clause Usage:

**COPYIN:** The COPYIN clause applies only to THREADPRIVATE common blocks. In the presence of the COPYIN clause, data from the master thread's copy of the common block is copied to the THREADPRIVATE copies upon entry to the parallel region.

**IF:** In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to `.TRUE.`. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

**NUM\_THREADS:** If the NUM\_THREADS clause is present, the corresponding expression, `scalar_integer_expression`, must evaluate to a positive integer value. This value sets the maximum number of threads used during execution of the parallel region. A NUM\_THREADS clause overrides either a previous call to the library routine `omp_set_num_threads()` or the setting of the `OMP_NUM_THREADS` environment variable.

## PARALLEL DO

The OpenMP PARALLEL DO directive is 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 (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.

### Syntax:

```

!$OMP PARALLEL DO [CLAUSES]
  < Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]

```

### Clauses:

PRIVATE(list)	COPYIN(list)
SHARED(list)	IF( <code>scalar_logical_expression</code> )
DEFAULT(PRIVATE   SHARED   NONE)	NUM_THREADS( <code>scalar_integer_expression</code> )
FIRSTPRIVATE(list)	SCHEDULE (type [, chunk])
LASTPRIVATE(list)	COLLAPSE (n)
REDUCTION( [{operator   intrinsic};] list)	ORDERED

**Usage:**

The semantics of the PARALLEL DO directive are identical to those of a parallel region containing only a single parallel DO loop and directive. The available clauses are the same as those defined in “[PARALLEL ... END PARALLEL](#),” on page 288 and “[DO...END DO](#)”.

**Note**


---

The END PARALLEL DO directive is optional.

## PARALLEL SECTIONS

The OpenMP PARALLEL SECTIONS / END SECTIONS directive pair define tasks to be executed in parallel; that is, they define a non-iterative work-sharing construct without the need to define an enclosing parallel region.

**Syntax:**

```
!$OMP PARALLEL SECTIONS [CLAUSES]
[!$OMP SECTION]
  < Fortran code block executed by processor i >
[!$OMP SECTION]
  < Fortran code block executed by processor j >
...
!$OMP END SECTIONS [NOWAIT]
```

**Clauses:**

PRIVATE(list)	REDUCTION({operator   intrinsic} : list)
SHARED(list)	COPYIN (list)
DEFAULT(PRIVATE   SHARED   NONE)	IF(scalar_logical_expression)
FIRSTPRIVATE(list)	NUM_THREADS(scalar_integer_expression)
LASTPRIVATE(list)	

**Usage:**

The PARALLEL SECTIONS / END SECTIONS directive pair define a non-iterative work-sharing construct without the need to define an enclosing parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing PARALLEL SECTIONS / END SECTIONS directives. In addition, the code within the PARALLEL SECTIONS / END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

Semantics are identical to a parallel region containing only an omp sections pragma and the associated structured block. The available clauses are as defined in “[PARALLEL ... END PARALLEL](#),” on page 288 and “[DO...END DO](#)”.

## PARALLEL WORKSHARE

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct.

**Syntax:**

```
!$OMP PARALLEL WORKSHARE [CLAUSES]
< Fortran structured block to be executed in parallel >
[!$OMP END PARALLEL WORKSHARE]
```

```
!$OMP PARALLEL DO [CLAUSES]
< Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

**Clauses:**

PRIVATE(list)	COPYIN (list)
SHARED(list)	IF(scalar_logical_expression)
DEFAULT(PRIVATE   SHARED   NONE)	NUM_THREADS(scalar_integer_expression)
FIRSTPRIVATE(list)	SCHEDULE (type [, chunk])
LASTPRIVATE(list)	COLLAPSE (n)
REDUCTION({operator   intrinsic} : list)	ORDERED

**Usage:**

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The semantics of the PARALLEL WORKSHARE directive are identical to those of a parallel region containing a single WORKSHARE construct.

The END PARALLEL WORKSHARE directive is optional, and NOWAIT may not be specified on an END PARALLEL WORKSHARE directive. The available clauses are as defined in “[PARALLEL ... END PARALLEL](#),” on page 288.

## SECTIONS ... END SECTIONS

The OpenMP SECTIONS / END SECTIONS directive pair define a non-iterative work-sharing construct within a parallel region in which each section is executed by a single processor.

**Syntax:**

```
!$OMP SECTIONS [ Clauses ]
[!$OMP SECTION]
  < Fortran code block executed by processor i >
[!$OMP SECTION]
  < Fortran code block executed by processor j >
  ...
!$OMP END SECTIONS [NOWAIT]
```

**Clauses:**

PRIVATE(list)	LASTPRIVATE(list)
FIRSTPRIVATE(list)	REDUCTION({operator   intrinsic} : list)

**Usage:**

The SECTIONS / END SECTIONS directive pair defines a non-iterative work-sharing construct within a parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors have no work and thus jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors must execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing SECTIONS / END SECTIONS directives. In addition, the code within the SECTIONS / END SECTIONS directives must be a structured block.

The available clauses are as defined in “PARALLEL ... END PARALLEL ,” on page 288 and “DO...END DO ”.

## SINGLE ... END SINGLE

The SINGLE...END SINGLE directive designates code that executes on a single thread and that is skipped by the other threads.

### Syntax:

```
!$OMP SINGLE [Clauses]
  < Fortran code executed in body of SINGLE processor section >
!$OMP END SINGLE [NOWAIT]
```

### Clauses:

PRIVATE(list)

FIRSTPRIVATE(list)

COPYPRIVATE(list)

### Usage:

In a parallel region of code, there may be a sub-region of code that only executes correctly on a single thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the SINGLE...END SINGLE directive pair lets you conveniently designate code that executes on a single thread and is skipped by the other threads.

The following restrictions apply to the SINGLE...END SINGLE directive:

- There is an implied barrier on exit from a SINGLE...END SINGLE section of code unless the optional NOWAIT clause is specified.
- Nested single process sections are ignored.
- Branching into or out of a single process section is not supported.

### Examples:

```
PROGRAM SINGLE_USE
  INTEGER A(0:1)
  INTEGER omp_get_thread_num()
!$OMP PARALLEL
  A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP SINGLE
  PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
  PRINT *, "A(0)=",A(0), " A(1)=", A(1)
END
```

## TASK

The OpenMP TASK directive defines an explicit task.

**Syntax:**

```
!$OMP TASK [Clauses]
  < Fortran code executed as task >
!$OMP END TASK
```

**Clauses:**

IF( <i>scalar_logical_expression</i> )	PRIVATE( <i>list</i> )
UNTIED	FIRSTPRIVATE( <i>list</i> )
DEFAULT( <i>private   firstprivate   shared   none</i> )	SHARED( <i>list</i> )

**Usage:**

The TASK / END TASK directive pair defines an explicit task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. The encountering thread may immediately execute the task, or delay its execution. If the task execution is delayed, then any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs.

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.

When an if clause is present on a task construct and the if clause expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs.

**Note**


---

Use of a variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the untied clause is present on a task construct, any thread in the team can resume the task region after a suspension.

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. An implementation may add task scheduling points anywhere in untied task regions.

**Note**


---

When storage is shared by an explicit task region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

**Restrictions:**

The following restrictions apply to the TASK directive:

- A program that branches into or out of a task region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the task directive, or on any side effects of the evaluations of the clauses.
- At most one *if* clause can appear on the directive.
- Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior.

## TASKWAIT

The OpenMP TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

### Syntax:

```
!$OMP TASKWAIT
```

### Clauses:

IF( <i>scalar_logical_expression</i> )	PRIVATE( <i>list</i> )
UNTIED	FIRSTPRIVATE( <i>list</i> )
DEFAULT( <i>private</i>   <i>firstprivate</i>   <i>shared</i>   <i>none</i> )	SHARED( <i>list</i> )

### Usage:

The OpenMP TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

### Restrictions:

The following restrictions apply to the TASKWAIT directive:

- The TASKWAIT directive and the `omp taskwait` pragma may be placed only at a point where a base language statement is allowed.
- The `taskwait` directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*.

## THREADPRIVATE

The OpenMP THREADPRIVATE directive identifies a Fortran common block as being private to each thread.

### Syntax:

```
!$OMP THREADPRIVATE (list)
```

### Usage:

The `list` for this directive is a comma-separated list of named variables to be made private to each thread or named common blocks to be made private to each thread but global within the thread.

On entry to a parallel region, data in a THREADPRIVATE common block or variable is undefined unless COPYIN is specified on the PARALLEL directive. When a common block or variable that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use.



**Restrictions:**

The following restrictions apply to the `THREADPRIVATE` directive:

- The `THREADPRIVATE` directive must appear after every declaration of a thread private common block.
- Only named common blocks can be made thread private.
- Common block names must appear between slashes, such as `/common_block_name/`.
- This directive must appear in the declarations section of a program unit after the declaration of any common blocks or variables listed.
- It is illegal for a `THREADPRIVATE` common block or its constituent variables to appear in any clause other than a `COPYIN` clause.
- A variable can appear in a `THREADPRIVATE` directive only in the scope in which it is declared. It must not be an element of a common block or be declared in an `EQUIVALENCE` statement.
- A variable that appears in a `THREADPRIVATE` directive and is not declared in the scope of a module must have the `SAVE` attribute.

**WORKSHARE ... END WORKSHARE**

The OpenMP `WORKSHARE ... END WORKSHARE` directive pair provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

**Syntax:**

```
!$OMP WORKSHARE
  < Fortran structured block to be executed in parallel >
!$OMP END WORKSHARE [NOWAIT]
```

**Usage:**

The Fortran structured block enclosed by the `WORKSHARE ... END WORKSHARE` directive pair can consist only of the following types of statements and constructs:

- Array assignments
- Scalar assignments
- `FORALL` statements or constructs
- `WHERE` statements or constructs
- OpenMP `ATOMIC`, `CRITICAL` or `PARALLEL` constructs

The work implied by these statements and constructs is split up between the threads executing the `WORKSHARE` construct in a way that is guaranteed to maintain standard Fortran semantics. The goal of the `WORKSHARE` construct is to effect parallel execution of non-iterative but implicitly data parallel array assignments, `FORALL`, and `WHERE` statements and constructs intrinsic to the Fortran language beginning with Fortran 90. The Fortran structured block contained within a `WORKSHARE` construct must not contain any user-defined function calls unless the function is `ELEMENTAL`.

## Runtime Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

### Note

Unlimited OpenMP thread counts are available in all PGI configurations. The number of threads is unlicensed in the OpenMP run-time libraries - up to the hard limit of 64 threads.

The following table summarizes the run-time library calls, providing an example for each.

Table 8.4. Run-time Library Routines Summary

Run-time Library Routines with Examples
<p><b>omp_get_num_threads</b></p> <p>Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region.</p> <p>By default, the value returned by this function is equal to the value of the environment variable <code>OMP_NUM_THREADS</code> or to the value set by the last previous call to <b>omp_set_num_threads()</b>.</p> <pre>integer function omp_get_num_threads();</pre>
<p><b>omp_set_num_threads</b></p> <p>Sets the number of threads to use for the next parallel region.</p> <p>This subroutine can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine that is called from within a parallel region, the results are undefined. Further, this subroutine has precedence over the <code>OMP_NUM_THREADS</code> environment variable.</p> <pre>subroutine omp_set_num_threads(scalar_integer_exp);</pre>
<p><b>omp_get_thread_num</b></p> <p>Returns the thread number within the team. The thread number lies between 0 and <b>omp_get_num_threads()-1</b>. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.</p> <pre>integer function omp_get_thread_num();</pre>
<p><b>omp_get_ancestor_thread_num</b></p> <p>Returns, for a given nested level of the current thread, the thread number of the ancestor.</p> <pre>integer function omp_get_ancestor_thread_num(level); integer level</pre>

Run-time Library Routines with Examples
<p><b>omp_get_active_level</b></p> <p>Returns the number of enclosing active parallel regions enclosing the task that contains the call. PGI currently supports only one level of active parallel regions, so the return value currently is 1.</p> <pre>integer function omp_get_active_level();</pre>
<p><b>omp_get_level</b></p> <p>Returns the number of parallel regions enclosing the task that contains the call.</p> <pre>integer function omp_get_level();</pre>
<p><b>omp_get_max_threads</b></p> <p>Returns the maximum value that can be returned by calls to <b>omp_get_num_threads()</b>.</p> <p>If <b>omp_set_num_threads()</b> is used to change the number of processors, subsequent calls to <b>omp_get_max_threads()</b> return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code.</p> <pre>integer function omp_get_max_threads();</pre>
<p><b>omp_get_num_procs</b></p> <p>Returns the number of processors that are available to the program.</p> <pre>integer function omp_get_num_procs();</pre>
<p><b>omp_get_stack_size</b></p> <p>Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread.</p> <p>This value may <i>not</i> be the size of the stack of the current thread.</p> <pre>!omp_get_stack_size interface function omp_get_stack_size () use omp_lib_kinds integer ( kind=OMP_STACK_SIZE_KIND ) :: omp_get_stack_size end function omp_get_stack_size end interface</pre>
<p><b>omp_set_stack_size</b></p> <p>Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread.</p> <p>The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the PGI implementation, all OpenMP or auto-parallelization threads are created just prior to the first parallel region; therefore, only calls to <b>omp_set_stack_size()</b> that occur prior to the first region have an effect.</p> <pre>subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND));</pre>

**Run-time Library Routines with Examples****omp\_get\_team\_size**

Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.

```
integer function omp_get_team_size (level)
integer level
```

**omp\_in\_parallel**

Returns whether or not the call is within a parallel region.

Returns `.TRUE.` if called from within a parallel region and `.FALSE.` if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.`, the function returns `.FALSE.`

```
logical function omp_in_parallel();
```

**omp\_set\_dynamic**

Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions.

This function is recognized, but currently has no effect.

```
subroutine omp_set_dynamic(scalar_logical_exp);
```

**omp\_get\_dynamic**

Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled.

This function is recognized, but currently always returns `.FALSE.`

```
logical function omp_get_dynamic();
```

**omp\_set\_nested**

Allows enabling/disabling of nested parallel regions.

This function is recognized, but currently has no effect.

```
subroutine omp_set_nested(nested);
  logical nested
```

**omp\_get\_nested**

Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled.

This function is recognized, but currently always returns `.FALSE.`

```
logical function omp_get_nested();
```

**Run-time Library Routines with Examples****omp\_set\_schedule**

Retrieve the value of the run\_sched\_var.

```
subroutine omp_set_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
```

**omp\_get\_schedule**

Retrieve the value of the run\_sched\_var.

```
subroutine omp_get_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
```

**omp\_get\_wtime**

Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value.

Times returned are per-thread times, and are not necessarily globally consistent across all threads.

```
double precision function omp_get_wtime();
```

**omp\_get\_wtick**

Returns the resolution of omp\_get\_wtime(), in seconds, as a DOUBLE PRECISION value.

```
double precision function omp_get_wtick();
```

**omp\_init\_lock**

Initializes a lock associated with the variable lock for use in subsequent calls to lock routines.

The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.

```
subroutine omp_init_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
```

**omp\_destroy\_lock**

Disassociates a lock associated with the variable.

```
subroutine omp_destroy_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
```

**Run-time Library Routines with Examples****omp\_set\_lock**

Causes the calling thread to wait until the specified lock is available.

The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.

```
subroutine omp_set_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
```

**omp\_unset\_lock**

Causes the calling thread to release ownership of the lock associated with `integer_var`.

If the variable is not already associated with a lock, it is illegal to make a call to this routine.

```
subroutine omp_unset_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
```

**omp\_test\_lock**

Causes the calling thread to try to gain ownership of the lock associated with the variable.

The function returns `.TRUE.` if the thread gains ownership of the lock; otherwise it returns `.FALSE.`

If the variable is not already associated with a lock, it is illegal to make a call to this routine.

```
logical function omp_test_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock
```

## OpenMP Environment Variables

OpenMP environment variables allow you to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives.

[Table 8.5, “OpenMP-related Environment Variable Summary Table,” on page 300](#) provides a brief summary of these variables. After the table this section contains more information about each of them. For complete information and more details related to these environment variables, refer to the OpenMP documentation available on the WorldWide Web.

Table 8.5. OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
<a href="#">OMP_DYNAMIC</a>	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
<a href="#">OMP_NESTED</a>		Currently has no effect. Typically specifies the maximum number of nested parallel regions.

Environment Variable	Default	Description
<code>OMP_MAX_ACTIVE_LEVELS</code>	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) nested parallelism.
<code>OMP_NUM_THREADS</code>	1	Specifies the number of threads to use during execution of parallel regions.
<code>OMP_SCHEDULE</code>	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for <i>omp for</i> and <i>omp parallel for</i> loops that include the run-time schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
<code>OMP_STACKSIZE</code>		Overrides the default stack size for a newly created thread.
<code>OMP_THREAD_LIMIT</code>	64	Specifies the absolute maximum number of threads that can be used in a program.
<code>OMP_WAIT_POLICY</code>	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

## OMP\_DYNAMIC

`OMP_DYNAMIC` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.

## OMP\_NESTED

`OMP_NESTED` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) nested parallelism.

## OMP\_MAX\_ACTIVE\_LEVELS

`OMP_MAX_ACTIVE_LEVELS` currently has no effect. Typically this variable specifies the maximum number of nested parallel regions. PGI ignores this variable value since nested parallelism is not supported.

## OMP\_NUM\_THREADS

`OMP_NUM_THREADS` specifies the number of threads to use during execution of parallel regions. The default value for this variable is 1. For historical reasons, the environment variable `NCPUS` is supported with the same functionality. In the event that both `OMP_NUM_THREADS` and `NCPUS` are defined, the value of `OMP_NUM_THREADS` takes precedence.

---

### Note

`OMP_NUM_THREADS` defines the threads that are used to execute the program, regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient.

## OMP\_SCHEDULE

`OMP_SCHEDULE` specifies the type of iteration scheduling to use for `DO` and `PARALLEL DO` loop directives that include the `SCHEDULE(RUNTIME)` clause, described in “[SCHEDULE,](#)” on page 280. The default value for this variable is `STATIC`.

If the optional chunk size is not set, a chunk size of 1 is assumed except in the case of a static schedule. For a static schedule, the default is as defined in “[DO...END DO ,](#)” on page 284.

Examples of the use of `OMP_SCHEDULE` are as follows:

```
% setenv OMP_SCHEDULE "STATIC, 5"
% setenv OMP_SCHEDULE "GUIDED, 8"
% setenv OMP_SCHEDULE "DYNAMIC"
```

## OMP\_STACKSIZE

`OMP_STACKSIZE` is an OpenMP 3.0 feature that controls the size of the stack for newly-created threads. This variable overrides the default stack size for a newly created thread. The value is a decimal integer followed by an optional letter B, K, M, or G, to specify bytes, kilobytes, megabytes, and gigabytes, respectively. If no letter is used, the default is kilobytes. There is no space between the value and the letter; for example, one megabyte is specified 1M. The following example specifies a stack size of 8 megabytes.

```
% setenv OMP_STACKSIZE 8M
```

The API functions related to `OMP_STACKSIZE` are `omp_set_stack_size` and `omp_get_stack_size`.

The environment variable `OMP_STACKSIZE` is read on program start-up. If the program changes its own environment, the variable is not re-checked.

This environment variable takes precedence over `MPSTKZ`, which increases the size of the stacks used by threads executing in parallel regions. Once a thread is created, its stack size cannot be changed.

In the PGI implementation, threads are created prior to the first parallel region and persist for the life of the program. The stack size of the main thread (thread 0) is set at program start-up and is not affected by `OMP_STACKSIZE`. For more information on controlling the program stack size in Linux, refer to “[Running Parallel Program on Linux](#)” in Chapter 2 of the PGI User’s Guide. For more information on `MPSTKZ`, refer to the PGI User’s Guide.

## OMP\_THREAD\_LIMIT

You can use the `OMP_THREAD_LIMIT` environment variable to specify the absolute maximum number of threads that can be used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using `set_omp_num_threads()`, cause the number of processes or threads to be set at the value of `OMP_THREAD_LIMIT` rather than the value specified in the function call.

## OMP\_WAIT\_POLICY

`OMP_WAIT_POLICY` sets the behavior of idle threads - specifically, whether they spin or sleep when idle. The values are `ACTIVE` and `PASSIVE`, with `ACTIVE` the default. The behavior defined by `OMP_WAIT_POLICY` is also shared by threads created by auto-parallelization.



- Threads are considered idle when waiting at a barrier, when waiting to enter a critical region, or when unemployed between parallel regions.
- Threads waiting for critical sections always busy wait (ACTIVE).
- Barriers always busy wait (ACTIVE), with calls to `sched_yield` determined by the environment variable `MP_SPIN`, that specifies the number of times it checks the semaphore before calling `sched_yield()` (on Linux, SUA, or MAC OS X) or `_sleep()` (on Windows).
- Unemployed threads during a serial region can either busy wait using the barrier (ACTIVE) or politely wait using a mutex (PASSIVE). This choice is set by `OMP_WAIT_POLICY`, so the default is ACTIVE.

When ACTIVE is set, idle threads consume 100% of their CPU allotment spinning in a busy loop waiting to restart in a parallel region. This mechanism allows for very quick entry into parallel regions, a condition which is good for programs that enter and leave parallel regions frequently.

When PASSIVE is set, idle threads wait on a mutex in the operating system and consume no CPU time until being restarted. Passive idle is best when a program has long periods of serial activity or when the program runs on a multi-user machine or otherwise shares CPU resources.



# Chapter 9. Interoperability with C

Fortran 2003 provides a mechanism for interoperating with C. Any entity involved must have equivalent declarations made in both C and Fortran. This section describes the pointer types and enumerators available for interoperability.

## Enumerators

Fortran 2003 has enumerators to provide interoperability with C. An enumerator is a set of integer constants. The kind of enumerator corresponds to the integer type that C would choose for the same set of constants.

You can specify the value of an enumerator when it is declared. If the value is not specified, then it is one of two values:

- If the enumerator is the first one, the value is zero.
- If the enumerator is not the first one, the value is one greater than the previous enumerator.

### Example 9.1. Enumerator Example

In the following example, green has a value of 4, purple a value of 8, and gold a value of 9.

```
enum, bind(c)
  enumerator :: green = 4, purple = 8
  enumerator gold
end enu
```

## Interoperability with C Pointer Types

C pointers are addresses. The derived type `c_ptr` is interoperable with C pointer types. The named constant `c_null_ptr` corresponds to the null value in C.

### `c_f_pointer`

A subroutine that assigns the C pointer target to the Fortran pointer, and specifies its shape.

F2003

Syntax

```
c_f_pointer (cptr, fptr [,shape] )
```

## Type

subroutine

## Description

`c_f_pointer` assigns the target, `cptr`, to the Fortran pointer, `fptr`, and specifies its shape.

- `cptr` is a scalar of the type `C_PTR` with `INTENT(IN)`. Its value is one of the following:
  - the C address of an interoperable data entity.

---

 Note

If `cptr` is the C address of a Fortran variable, the Fortran variable must have the `target` attribute.

- the result of a reference to `c_loc` with a non-interoperable argument.
- `fptr` is a pointer with `INTENT(OUT)`.
- If `cptr` is the C address of an interoperable data entity, then `fptr` must be a data pointer of the type and type parameters of the entity. It becomes pointer associated with the target of `cptr`.
- If `cptr` was returned by a call of `c_loc` with a non-interoperable argument `x`, then `fptr` must be a nonpolymorphic scalar pointer of the type and type parameters of `x`.

---

 Note

`x` or its target if it is a pointer, must not have been deallocated or become undefined due to execution of a `return` or `end` statement.

`fptr` is associated with `x` or its target.

- `shape` is an optional argument that is a rank-one array of type `INTEGER` with `INTENT(IN)`. `shape` is present if and only if `fptr` is an array. The size must be equal to the rank of `fptr`; each lower bound is assumed to be 1.

## Example

```

program main
  use iso_c_binding
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')
      import :: c_ptr
      type(c_ptr), intent(out) :: p
    end subroutine
  end interface
  type(c_ptr) :: cptr
  real,pointer :: a(:)
  call my_routine(cptr)
  call c_f_pointer(cptr, a, [12])
end program main

```

## c\_f\_procpointer

A subroutine that associates a procedure pointer with the target of a C function pointer.

F2003

### Syntax

```
c_f_procpointer (cptr, fptr )
```

### Type

subroutine

### Description

`c_f_procpointer` associates a procedure pointer with the target of a C function pointer.

- `cptr` is a scalar of the type `C_PTR` with `INTENT(IN)`. Its value is the C address of the procedure that is interoperable.

Its value is one of the following:

- the C address of an interoperable procedure.
- the result of a reference to `c_loc` with a non-interoperable argument. In this case, there is no intent that any use of it be made within C except to pass it back to Fortran, where `C_F_POINTER` is available to reconstruct the Fortran pointer.
- `fptr` is a procedure pointer with `INTENT(OUT)`.

The interface for `fptr` shall be interoperable with the prototype that describes the target of `cptr`.

`fptr` becomes pointer associated with the target of `cptr`

- If `cptr` is the C address of an interoperable procedure, then the interface for `fptr` shall be interoperable with the prototype that describes the target of `cptr`. `fptr` must be a data pointer of the type and type parameters of the entity. It becomes pointer associated with the target of `cptr`.
- If `cptr` was returned by a call of `c_loc` with a non-interoperable argument `x`, then `fptr` must be a nonpolymorphic scalar pointer of the type and type parameters of `x`.

### Note

`x` or its target if it is a pointer, must not have been deallocated or become undefined due to execution of a `return` or `end` statement.

`fptr` is associated with `x` or its target.

### Example

```

program main
  use iso_c_binding
  implicit none
  interface
    subroutine my_routine(p) bind(c,name='myC_func')

```

```
import :: c_ptr
type(c_ptr), intent(out) :: p
end subroutine
end interface
type(c_ptr) :: cptr
real, pointer :: a(:)
call my_routine(cptr)
call c_f_pointer(cptr, a, [12])
end program main
```

### c\_associated

A subroutine that determines the status of a C\_PTR, or determines if one C\_PTR is associated with a target C\_PTR.

F2003

#### Syntax

```
c_associated (cptr1[, cptr2] )
```

#### Type

subroutine

#### Description

c\_associated determines the status of a C\_PTR, cptr1, or determines if cptr1 is associated with a target cptr2.

- cptr1 is a scalar of the type C\_PTR.
- cptr2 is an optional scalar or the same type as cptr1.

#### Return Value

A logical value:

- .false. if either cptr1 is a C NULL pointer or if cptr1 and cptr2 point to different addresses.
- .true. if cptr1 is a not a C NULL pointer or if cptr1 and cptr2 point to the same address.

#### Example

```
program main
  use iso_c_binding
  subroutine test_association(h,k)
    only: c_associated, c_loc, c_ptr
    real, pointer :: h
    type(c_ptr) :: k
    if(c_associated(k, c_loc(h))) &
      stop 'h and k do not point to same target'
  end subroutine test_association
```

## Interoperability of Derived Types

For a derived type to be interoperable, the following must be true:

- It must have the `bind` attribute.

```
type, bind(c) :: atype
:
end type atype
```

- It cannot be a sequence type.
- It cannot have type parameters.
- It cannot have the `extends` attribute.
- It cannot have any type-bound procedures.
- Each component must comply with these rules:
  - Must have interoperable type and type parameters.
  - Must not be a pointer.
  - Must not be allocatable.

Under the preceding conditions the type can interoperate with a C struct type that has the same number of components, with components corresponding by their position in the definitions. Further, each Fortran component must be interoperable with its corresponding C component. The name of the type and names of the components is not significant for interoperability.

There is no Fortran type that is interoperable with these C types:

- a C union type,
- a C struct type that contains a bit field
- a C struct type that contains a flexible array member.

### Example 9.2. Derived Type Interoperability

This type...	Is interoperable with this type
<pre>typedef struct {   int a,b;   float t; } my_c_type</pre>	<pre>use iso_c_binding type, bind(c) :: my_fort_type   integer(c_int) :: i,j   real(c_float) :: s end type my_fort_type</pre>





# Chapter 10. HPF

This chapter contains information about HPF, High Performance Fortran. HPF is an extension of Fortran 90 with constructs that support parallel computing.

## Supported HPF Intrinsic

The following table lists the HPF intrinsics and Library procedures supported by the PGHPF compiler. Refer to the man pages supplied with the PGHPF software for further details on these intrinsics and procedures. Refer to [Chapter 10, “HPF”](#), for the HPF\_LIBRARY\_LOCAL routines.

Table 10.1. HPF Intrinsic and Library Procedures

Intrinsic	Class
ALL_PREFIX	Transformational function
ALL_SCATTER	Transformational function
ALL_SUFFIX	Transformational function
ANY_PREFIX	Transformational function
ANY_SCATTER	Transformational function
ANY_SUFFIX	Transformational function
COPY_PREFIX	Transformational function
COPY_SCATTER	Transformational function
COPY_SUFFIX	Transformational function
COUNT_PREFIX	Transformational function
COUNT_SCATTER	Transformational function
COUNT_SUFFIX	Transformational function
GRADE_DOWN	Transformational function
GRADE_UP	Transformational function
HPF_ALIGNMENT	Mapping inquiry subroutine
HPF_DISTRIBUTION	Mapping inquiry subroutine

<b>Intrinsic</b>	<b>Class</b>
HPF_TEMPLATE	Mapping inquiry subroutine
IALL	Transformational function
IALL_PREFIX	Transformational function
IALL_SCATTER	Transformational function
IALL_SUFFIX	Transformational function
IANY	Transformational function
IANY_PREFIX	Transformational function
IANY_SCATTER	Transformational function
IANY_SUFFIX	Transformational function
ILEN	Elemental Intrinsic
IPARITY	Transformational function
IPARITY_PREFIX	Transformational function
IPARITY_SCATTER	Transformational function
IPARITY_SUFFIX	Transformational function
LEADZ	Elemental function
MAXLOC	Transformational function Intrinsic
MAXVAL_PREFIX	Transformational function
MAXVAL_SCATTER	Transformational function
MAXVAL_SUFFIX	Transformational function
MINLOC	Transformational function Intrinsic
MINVAL_PREFIX	Transformational function
MINVAL_SCATTER	Transformational function
MINVAL_SUFFIX	Transformational function
NUMBER_OF_PROCESSORS	System Inquiry function Intrinsic
PARITY	Transformational function
PARITY_PREFIX	Transformational function
PARITY_SCATTER	Transformational function
PARITY_SUFFIX	Transformational function
POPCNT	Elemental function
POPPAR	Elemental function
PROCESSORS_SHAPE	System Inquiry function Intrinsic
PRODUCT_PREFIX	Transformational function
PRODUCT_SCATTER	Transformational function

Intrinsic	Class
PRODUCT_SUFFIX	Transformational function
SUM_PREFIX	Transformational function
SUM_SCATTER	Transformational function
SUM_SUFFIX	Transformational function

## CM Fortran Intrinsics

This section provides information on CM Fortran intrinsics. The PGHPF compiler option `-Mcmf` provides limited support for CM Fortran compatibility (Thinking Machines Corporation version of Fortran). This includes support for the intrinsics `DOTPRODUCT`, `DLBOUND`, `DUBOUND`, and `DSHAPE` which have calling sequences identical to their Fortran 90/95 counterparts. It also includes support for the CM Fortran method of using square brackets in the definition of array constructors and the use of the `ARRAY` keyword in place of the Fortran 90/95 standard `DIMENSION` keyword.

There are three CM Fortran intrinsics which have names identical to their Fortran 90/95 counterparts but whose calling sequences differ. These are `CSHIFT`, `EOSHIFT`, and `RESHAPE`; their descriptions follow.

If PGHPF is invoked with the compiler switch `-Mcmf` these three intrinsics will be interpreted using the CM Fortran convention rather than the standard Fortran 90/95 convention. There are 6 additional non-standard intrinsics in CM Fortran: `PROJECT`, `LASTLOC`, `FIRSTLOC`, `RANK`, `DIAGONAL`, and `REPLICATE`. These non-standard intrinsics are not supported by PGHPF. Other features of CM Fortran that are not supported are the layout directives and the utility routines.

### CSHIFT

Perform a circular shift on the specified array.

### CMF

### Synopsis

```
CSHIFT(ARRAY, DIM, SHIFT)
```

### Arguments

The argument `ARRAY` is the array to shift. It may be an array of any type. The argument `DIM` is an integer representing the dimension to shift. The argument `SHIFT` is an integer or an array of integers with rank `n-1` where `n` is the rank of `ARRAY`.

### Return Value

The shifted array with the same size and shape as the argument `ARRAY`.

### EOSHIFT

Perform an end-off shift on the specified array.

## CMF

## Synopsis

```
CSHIFT(ARRAY, DIM, SHIFT, BOUNDARY)
```

## Arguments

The argument `ARRAY` is the array to shift. It may be an array of any type. The argument `DIM` is an integer representing the dimension to shift. The argument `SHIFT` is an integer or an array of integers with rank `n-1` where `n` is the rank of `ARRAY`. The optional argument `BOUNDARY` is of the same type as the array, it may be scalar or of rank `n-1` where `n` is the rank of `ARRAY`. `BOUNDARY` is the value to fill in the shifted out positions. By default it has the following values for integer, 0, for real, 0.0, for complex, (0.0,0.0), for logical false, for character the default is blank characters.

## Return Value

The shifted array with the same size and shape as the argument `ARRAY`.

## RESHAPE

Reconstructs an array with the specified shape using the elements of the source array.

## CMF

## Synopsis

```
RESHAPE(SHAPE, SOURCE [,PAD] [,ORDER])
```

## Arguments

The argument `SHAPE` is of type integer, rank one. It must not have more than 7 elements and no values can be negative. The argument `SOURCE` is an array of any type. The optional argument `PAD` must be the same size and type as `SOURCE`. The optional argument `ORDER` must be of type integer and must have the same shape as `SHAPE`.

## Return Value

The return value is an array of shape `SHAPE`, with the same type as `SOURCE`. Array elements are filled into the new array in array element order.

## HPF Directives

HPF directives are Fortran 90/95 comments which convey information to the PGHPF compiler. Directives are the heart of an HPF program, indicating data parallelism by specifying how data is assigned and allocated among processors on a parallel system, and the interrelationships between various data elements.

### Adding HPF Directives to Programs

Directives in an HPF program may have any of the following forms:

```
CHPF$ directive
!HPF$ directive
*HPF$ directive
```

Since HPF supports two source forms, fixed source form, and free source form, there are a variety of methods to enter a directive. Section 3.4 of the Fortran 95 Handbook outlines methods for entering code that is valid for both free and fixed form Fortran. The C, !, or \* must be in column 1 for fixed source form directives. In free source form, Fortran limits the comment character to !. If you use the !HPF\$ form for the directive origin, and follow the rules outlined in the Fortran 95 Handbook, your code will be universally valid. The body of the directive may immediately follow the directive origin. Alternatively, using free source form, any number of blanks may precede the HPF directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equals sign.

The directive name, including the directive origin, may contain upper or lower case letters (case is not significant).

## HPF Directive Summary

Table 10.2. HPF Directive Summary

DIRECTIVE	FUNCTION
ALIGN	Specifies that a data object is mapped in the same fashion as an associated data object. This is a specification statement. By default, objects are aligned to themselves.
DIMENSION	Specifies the dimensions of a template or processor "array". This is a specification statement.
DISTRIBUTE	Specifies the mapping of data objects to processors. This is a specification statement. By default, objects are not distributed.
DYNAMIC	Specifies that an object may be dynamically realigned or redistributed.
INDEPENDENT	Preceding a DO loop or FORALL, this directive specifies that the DO loop's iterations do not interact in any way and that the FORALL index computations do not interfere with each other, and thus the FORALL may be executed in parallel. This is an executable statement. By default, FORALL and DO loops are not assumed to be independent.
INHERIT	Specifies that a subprogram's dummy argument use the template associated with the actual argument for its alignment. This is a specification statement.
NOSEQUENCE	Specifies variables that are not sequential. Note that using PGHPF, by default variables is not sequential. Variables will be sequential if the compiler option -Msequence is supplied.
PROCESSORS	Specifies the number and rank of a processor arrangement. This is a specification statement.
REALIGN	This is similar to ALIGN, but is executable. An array can be realigned at any time, if it is declared using the DYNAMIC attribute.

DIRECTIVE	FUNCTION
REDISTRIBUTE	This is similar to DISTRIBUTE, but is executable. An array can be redistributed at any time, if it is declared using the DYNAMIC attribute.
SEQUENCE	Specifies that a variable or common block is sequential and requires linear, standard FORTRAN 77, treatment. This is a specification statement.
TEMPLATE	Defines an entity that may be used as an abstract align-target for a distribution or a redistribution. This is a specification statement.

## ALIGN - REALIGN

The ALIGN directive specifies how data objects are mapped in relation to other data objects. The data objects that are most often aligned in HPF programs are arrays. Alignment suggests to the compiler that entire objects or elements of arrays be stored on the same processor. Operations on objects that are aligned should be more efficient than operations on objects that are not aligned, assuming that objects that are not aligned may reside on different processors.

REALIGN is similar to ALIGN, but is executable. An array can be realigned at any time, if it is declared using the DYNAMIC attribute.

### Syntax

```
!HPF$ ALIGN alignee align-directive-stuff
```

or

```
!HPF$ ALIGN align-attribute-stuff :: alignee-list
```

where:

**alignee**

is an object-name.

**align-directive-stuff**

is (align-source-list) align -with-clause

**align-attribute-stuff**

is [(align-source-list)] align -with-clause

Each align-source has the form:

```
:
*
align-dummy
```

Each align-with-clause has the form:

```
WITH align-target [ ( align-subscript-list ) ]
```

An align-subscript has the form:

```
int-exp
align-subscript-use
```

```
subscript-triplet
*
```

## Type

### Specification

### Default

The default PGHPF alignment specifies that a data object is replicated across all processor memories. For example, for an array `RAY1` with a single dimension and a template `T` with matching size and shape, the following alignment specifies replication when `T` is distributed in any manner across processors.

```
!HPF$ ALIGN RAY1(*) WITH T(*)
!HPF$ DISTRIBUTE T(BLOCK)
```

### See Also

For details on the `ALIGN` syntax specifications, refer to the sections on `Align` and `Realign` in either *The High Performance Fortran Handbook* or in the *HPF Language Specification*.

### Example

```
PROGRAM TEST
INTEGER A(1000)
!HPF$ PROCESSORS PROC(10)
!HPF$ TEMPLATE T(1000)
!HPF$ ALIGN A(:) WITH T(:)
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC:: T
```

## DIMENSION

The `DIMENSION` attribute specifies the dimensions and extents for each dimension of a `TEMPLATE` or `PROCESSORS` directive.

### Syntax

```
!HPF$ DIMENSION ( explicit-shape-spec-list )
```

## Type

### Specification

### Default

The default for a `TEMPLATE` or `PROCESSORS` arrangement is a scalar.

### See Also

[“TEMPLATE”](#) and [“PROCESSORS”](#) directives

### Example

```
REAL A(100,100)
!HPF$ PROCESSORS, DIMENSION(10,10):: PROC
```

```
!HPF$ TEMPLATE, DIMENSION(10,10):: T
!HPF$ ALIGN WITH T:: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

## DYNAMIC

The **DYNAMIC** attribute specifies that an object may be dynamically realigned or redistributed.

### Syntax

```
!HPF$ DYNAMIC alignee-or-distributeee-list
```

### Type

Specification

### Default

By default an object is not dynamic.

### See Also

[“ALIGN - REALIGN”](#) and [“DISTRIBUTE - REDISTRIBUTE ”](#) directives

### Example

```
REAL A(100,100)
!HPF$ DYNAMIC A
!HPF$ PROCESSORS, DIMENSION(10,10):: PROC
!HPF$ TEMPLATE, DIMENSION(10,10):: T
!HPF$ ALIGN WITH T:: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

## DISTRIBUTE - REDISTRIBUTE

The **DISTRIBUTE** directive specifies a mapping of data objects to abstract processors in a processor arrangement. Distribution partitions an object, in the usual case an array (actually a template), among a set of processors.

**REDISTRIBUTE** is similar to **DISTRIBUTE**, but is executable. An array can be redistributed at any time, if it is declared using the **DYNAMIC** attribute

### Syntax

```
!HPF$ DISTRIBUTE distributeee dist-directive-stuff
```

or

```
!HPF$ DISTRIBUTE dist-attribute-stuff :: distributeee-list
```

where **dist-directive-stuff** is one of:

```
(dist-format-list)
(dist-format-list) ONTO processors-name
```

The form of **dist-attribute-stuff** is one of:



```
(dist-format-list)
(dist-format-list) ONTO processors-name
ONTO dist-target
```

The dist-format may be one of:

```
BLOCK [ (int-expr) ]
CYCLIC [ (int-expr) ]
```

## Type

### Specification

### Default

By default, each object is replicated and distributed to every processor.

### See Also

For details on the DISTRIBUTE syntax specifications, refer to the sections on DISTRIBUTE and REDICTRIBUTE in The High Performance Fortran Handbook or in the HPF Language Specification.

### Example

```
REAL A(100,100)
!HPF$ PROCESSORS PROC(10,10)
!HPF$ TEMPLATE T(10,10)
!HPF$ ALIGN WITH T:: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

## INDEPENDENT

The INDEPENDENT directive specifies that the iterations of a DO loop, or the computations for the active index values of a FORALL, do not interfere with each other in any way. Refer to the PGHPF Release notes for details on extensions to the INDEPENDENT directive.

## Syntax

```
!HPF$ INDEPENDENT [, NEW ( variable-list
) ]
```

## Type

Executable

### Default

By default, DO and FORALL statements are not independent.

### See Also

For details on the INDEPENDENT syntax specifications, refer either to Section 6.4 of The High Performance Fortran Handbook, or Section 4.4 of the HPF Language Specification. Also refer to the PGHPF Release notes for details on extensions to the INDEPENDENT directive.

## Example

```
!HPF$ INDEPENDENT
DO I = 2, N-1
X(I) = Y(I-1) + Y(I) + Y(I+1)
END DO
```

## INHERIT

The **INHERIT** directive specifies that the template for a dummy argument should be the same as the template for the corresponding actual argument.

### Syntax

```
!HPF$ INHERIT dummy-argument-name-list
```

### Default

If the **INHERIT** attribute is not used, and **ALIGN** and **DISTRIBUTE** are not used for a dummy argument, then the dummy's template has the same shape as the dummy argument and it is ultimately aligned with itself.

### Type

Specification

### See Also

For details on the **INHERIT** syntax specifications, refer either to Section 5.4 of *The High Performance Fortran Handbook*, or Section 3.9 of the *HPF Language Specification*.

## Example

```
REAL VAR1(100)
!HPF$ DISTRIBUTE VAR1(BLOCK)10))
CALL SUB1( VAR1(10:20:2))
SUBROUTINE SUB1(PARAM1)
REAL PARAM1(5)
!HPF$ INHERIT PARAM1
```

## PROCESSORS

The **PROCESSORS** directive specifies one or more processor arrangements, by name, rank, and size.

### Syntax

```
!HPF$ PROCESSORS processors-decl-list
```

### Default

The default for **PROCESSORS** is the number of processors on which the program is running, as specified by the runtime command-line options.

### Type

Specification

## See Also

For details on the PROCESSOR syntax specifications, refer either to Section 4.8 of The High Performance Fortran Handbook, or Section 3.7 of the HPF Language Specification

For finding more information on processors while running a program, refer to the NUMBER\_OF\_PROCESSORS and PROCESSORS\_SHAPE intrinsics.

## Examples

```
!HPF$ PROCESSORS PROCN(128)
!HPF$ PROCESSORS PROC2(3,3,3)
!HPF$ PROCESSORS:: PROC3(-8:12,100:200)
```

## NO SEQUENCE

In environments where variables are by default sequential, the NO SEQUENCE directive specifies that non-sequential access should apply to a scoping unit or to variables and common blocks within the scoping unit.

## Syntax

```
!HPF$ NO SEQUENCE
```

or

```
!HPF$ NOSEQUENCE [::] association-name-list
```

## Type

Specification

## See Also

For details on the NO SEQUENCE syntax specifications, refer either to Section 4.10.2 of The High Performance Fortran Handbook, or Section 7.1.3 of the HPF Language Specification

The SEQUENCE directive.

## Example

```
INTEGER FLAG, I, A(1000)
COMMON /FOO/ A,I,FLAG
!HPF$ NOSEQUENCE FOO
```

## SEQUENCE

The SEQUENCE directive allows a user to declare explicitly that variables or common blocks are to be treated by the compiler as sequential.

## Syntax

```
!HPF$ SEQUENCE
```

or

## HPF\_LOCAL

```
!HPF$ SEQUENCE [::] association-name-list
```

### Type

Specification

### See Also

For details on the SEQUENCE syntax specifications, refer either to Section 4.10.2 of The High Performance Fortran Handbook, or Section 7.1.3 of the HPF Language Specification.

The NO SEQUENCE directive.

### Example

```
INTEGER FLAG, I, A(1000)
COMMON /FOO/ A,I,FLAG
!HPF$ SEQUENCE FOO
```

## TEMPLATE

The TEMPLATE directive declares one or more templates, specifying for each a name, rank, and size for each dimension.

### Syntax

```
!HPF$ TEMPLATE template-decl-list
```

### Default

By default for each object, a new template is created and in the absence of an explicit ALIGN directive, the object is ultimately aligned to itself.

### Type

Specification

### See Also

For details on the TEMPLATE syntax specifications, refer either to Section 4.9 of The High Performance Fortran Handbook, or Section 3.8 of the HPF Language Specification.

### Examples

```
!HPF$ TEMPLATE VAR1(N)
!HPF$ TEMPLATE VAR2(N,N)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK):: BOARD(8,8)
```

## HPF\_LOCAL

This appendix lists the HPF\_LOCAL\_LIBRARY procedures. For complete descriptions of the HPF\_LOCAL\_LIBRARY routines, and the current standards for HPF\_LOCAL extrinsics, refer to Annex A, "Coding

Local Routines in HPF and Fortran 90", in the High Performance Fortran Language Specification. [Table 10.3, "HPF\\_LOCAL\\_LIBRARY Procedures"](#), briefly lists the procedures. Refer to the man pages supplied with the PGHPF software for further details on these procedures. Refer to [Chapter 6, "Fortran Ininsics"](#), for details on the intrinsics defined in the Fortran 90/95 Language Specification and for HPF LIBRARY procedures.

For complete descriptions of the HPF\_LOCAL\_LIBRARY routines, and the current standards for HPF\_LOCAL extrinsics, refer to Annex A, "Coding Local Routines in HPF and Fortran 90", in the High Performance Fortran Language Specification.

Table 10.3. HPF\_LOCAL\_LIBRARY Procedures

Intrinsic	Description
ABSTRACT_TO_PHYSICAL	Returns processor identification for the physical processor associated with a specified abstract processor.
GLOBAL_ALIGNMENT	Returns information about the global HPF array argument.
GLOBAL_DISTRIBUTION	Returns information about the global HPF array argument.
GLOBAL_LBOUND	Returns lower bounds of the actual HPF global array associated with a dummy array.
GLOBAL_SHAPE	Returns the shape of the global HPF actual argument.
GLOBAL_SIZE	Returns the global extent of the specified argument.
GLOBAL_TEMPLATE	Returns template information for the global HPF array argument.
GLOBAL_TO_LOCAL	Converts a set of global coordinates within a global HPF actual argument.
GLOBAL_UBOUND	Returns upper bounds of the actual HPF global array associated with a dummy array.
LOCAL_BLKCNT	Returns the number of blocks of elements in each dimension on a given processor.
LOCAL_LINDEX	Returns the lowest local index of all blocks of an array dummy.
LOCAL_TO_GLOBAL	Converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates.
LOCAL_UINDEX	Returns the highest local index of all blocks of an array dummy argument.
MY_PROCESSOR	Returns the identifying number of the calling physical processor.

Intrinsic	Description
PHYSICAL_TO_ABSTRACT	Returns coordinates for an abstract processor, relative to a global actual argument array.

## ABSTRACT\_TO\_PHYSICAL

Subroutine returns processor identification for the physical processor associated with a specified abstract processor relative to a global actual argument array.

### Synopsis

```
ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)
```

### Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

INDEX must be a rank-1 integer array containing the coordinates of an abstract processor in the processors arrangement onto which the global HPF array is mapped. It is an INTENT(IN) argument. The size of INDEX must equal the rank of the processors arrangement.

PROC must be scalar and of type integer. It is an INTENT(OUT) argument. It receives the identifying value for the physical processor associated with the abstract processor specified by INDEX.

## GLOBAL\_ALIGNMENT

This has the same interface and behavior as the HPF inquiry subroutine HPF\_ALIGNMENT, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

### Synopsis

```
GLOBAL_ALIGNMENT(ARRAY, ...)
```

## GLOBAL\_DISTRIBUTION

This has the same interface and behavior as the HPF inquiry subroutine HPF\_DISTRIBUTION, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

### Synopsis

```
GLOBAL_DISTRIBUTION(ARRAY, ...)
```

## GLOBAL\_LBOUND

Inquiry function, returns all the lower bounds or a specified lower bound of the actual HPF global array.

### Synopsis

```
GLOBAL_LBOUND(ARRAY, DIM)
```

## Arguments

Optional argument. DIM

ARRAY may be of any type. It must not be a scalar. It must be a dummy array argument of an HPF\_LOCAL procedure which is argument associated with a global HPF array actual argument.

DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

## Return Type

The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size  $n$ , where  $n$  is the rank of ARRAY.

## Return Value

If the actual argument associated with the actual argument associated with ARRAY is an array section or an array expression, other than a whole array or an array structure component, GLOBAL\_LBOUND(ARRAY, DIM) has the value 1; otherwise it has a value equal to the lower bound for subscript DIM of the actual argument associated with the actual argument associated with ARRAY.

GLOBAL\_LBOUND(ARRAY) has a value whose  $i$ th component is equal to GLOBAL\_LBOUND(ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$  where  $n$  is the rank of ARRAY.

## GLOBAL\_SHAPE

Returns the shape of the global HPF actual argument associated with an array or scalar dummy argument of an HPF\_LOCAL procedure.

## Synopsis

```
GLOBAL_SHAPE ( SOURCE )
```

## Argument

SOURCE may be of any type. It may be array valued or a scalar. It must be a dummy argument of an HPF\_LOCAL procedure which is argument associated with a global HPF actual argument.

## Return Type

The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

## Return Value

The value of the result is the shape of the global actual argument associated with the actual argument associated with SOURCE.

## GLOBAL\_SIZE

Inquiry function returns the extent along a specified dimension of the global HPF actual array argument associated with a dummy array argument of an HPF\_LOCAL procedure.

## Synopsis

```
GLOBAL_SIZE(ARRAY, DIM)
```

## Arguments

ARRAY may be of any type. It must not be a scalar. It must be a dummy argument of an HPF\_LOCAL procedure which is argument associated with a global HPF actual argument.

DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY.

## Return Type

Default integer scalar.

## Return Value

The result has a value equal to the extent of dimension DIM of the actual argument associated with the actual argument associated with ARRAY or, if DIM is absent, the total number of elements in the actual argument associated with the actual argument associated with ARRAY.

## GLOBAL\_TEMPLATE

This has the same interface and behavior as the HPF inquiry subroutine HPF\_TEMPLATE, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

## Synopsis

```
GLOBAL_TEMPLATE(ARRAY, ...)
```

## Arguments

Refer to HPF\_TEMPLATE.

## GLOBAL\_TO\_LOCAL

Subroutine converts a set of global coordinates within a global HPF actual argument array to an equivalent set of local coordinates within the associated local dummy array.

## Synopsis

```
GLOBAL_TO_LOCAL(ARRAY, G_INDEX,  
L_INDEX, LOCAL,  
NCOPIES, PROCS)
```

## Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.



`G_INDEX` must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(IN)` argument. It contains the coordinates of an element within the global HPF array actual argument associated with the local dummy array `ARRAY`.

`L_INDEX` (optional) must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(OUT)` argument. It receives the coordinates within a local dummy array of the element identified within the global actual argument array by `G_INDEX`. (These coordinates are identical on any processor that holds a copy of the identified global array element.)

`LOCAL` (optional) must be scalar and of type `LOGICAL`. It is an `INTENT(OUT)` argument. It is set to `.TRUE.` if the local array contains a copy of the global array element and to `.FALSE.` otherwise.

`NCOPIES` (optional) must be scalar and of type integer. It is an `INTENT(OUT)` argument. It is set to the number of processors that hold a copy of the identified element of the global actual array.

`PROCS` (optional) must be a rank-1 integer array whose size is at least the number of processors that hold copies of the identified element of the global actual array. The identifying numbers of those processors are placed in `PROCS`. The order in which they appear is implementation dependent.

## GLOBAL\_UBOUND

Inquiry function returns all the upper bounds or a specified upper bound of the actual HPF global array argument associated with an `HPF_LOCAL` dummy array argument.

### Synopsis

```
GLOBAL_UBOUND ( ARRAY , DIM )
```

### Arguments

Optional argument. `DIM`

`ARRAY` may be of any type. It must not be a scalar. It must be a dummy array argument of an `HPF_LOCAL` procedure which is argument associated with a global HPF array actual argument.

`DIM` (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

### Return Type

The result is of type default integer. It is scalar if `DIM` is present; otherwise the result is an array of rank one and size  $n$ , where  $n$  is the rank of `ARRAY`.

### Return Value

If the actual argument associated with the actual argument associated with `ARRAY` is an array section or an array expression, other than a whole array or an array structure component, `GLOBAL_UBOUND(ARRAY, DIM)` has a value equal to the number of elements in the given dimension; otherwise it has a value equal to the upper bound for subscript `DIM` of the actual argument associated with the actual argument associated with `ARRAY`, if dimension `DIM` does not have size zero and has the value zero if dimension `DIM` has size zero.

`GLOBAL_UBOUND(ARRAY)` has a value whose  $i$ th component is equal to `GLOBAL_UBOUND(ARRAY, i)`, for  $i = 1, 2, \dots, n$  where  $n$  is the rank of `ARRAY`.

## LOCAL\_BLKCNT

Pure function returns the number of blocks of elements in each dimension, or of a specific dimension of the array on a given processor.

### Synopsis

```
LOCAL_BLKCNT(ARRAY, DIM, PROC)
```

### Arguments

Optional arguments. DIM, PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM (optional) must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$  where  $n$  is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

### Return Type

The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size  $n$ , where  $n$  is the rank of ARRAY.

### Return Value

The value of LOCAL\_BLKCNT(ARRAY, DIM, PROC) is the number of blocks of the ultimate align target of ARRAY in dimension DIM that are mapped to processor PROC and which have at least one element of ARRAY aligned to them.

LOCAL\_BLKCNT(ARRAY, DIM) returns the same value as LOCAL\_BLKCNT(ARRAY, DIM, PROC=MY\_PROCESSOR()).

LOCAL\_BLKCNT(ARRAY) has a value whose  $i$ th component is equal to LOCAL\_BLKCNT(ARRAY,  $i$ ), for  $i = 1, \dots, n$ , where  $n$  is the rank of ARRAY.

## LOCAL\_LINDEX

Pure function returns the lowest local index of all blocks of an array dummy argument in a given dimension on a processor.

### Synopsis

```
LOCAL_LINDEX(ARRAY, DIM, PROC)
```

### Arguments

Optional argument. PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

### Return Type

The result is a rank-one array of type default integer and size  $1 \leq \text{DIM} \leq n$ , where  $n$  is the value returned by LOCAL\_BLKCNT(ARRAY, DIM [, PROC]).

### Return Value

The value of LOCAL\_LINDEX(ARRAY, DIM, PROC) has a value whose  $i$ th component is the local index of the first element of the  $i$ th block in dimension DIM of ARRAY on processor PROC.

LOCAL\_LINDEX(ARRAY, DIM) returns the same value as LOCAL\_LINDEX(ARRAY, DIM, PROC=MY\_PROCESSOR()).

## LOCAL\_TO\_GLOBAL

Subroutine converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates within the associated global HPF actual argument array.

### Synopsis

```
LOCAL_TO_GLOBAL ( ARRAY , L_INDEX ,
                 G_INDEX )
```

### Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

L\_INDEX must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(IN) argument. It contains the coordinates of an element within the local dummy array ARRAY.

G\_INDEX must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(OUT) argument. It receives the coordinates within the global HPF array actual argument of the element identified within the local array by L\_INDEX.

## LOCAL\_UINDEX

Pure function returns the highest local index of all blocks of an array dummy argument in a given dimension on a processor.

### Synopsis

```
LOCAL_UINDEX ( ARRAY , DIM , PROC )
```

### Arguments

Optional argument. PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM must be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

### Return Type

The result is a rank-one array of type default integer and size  $b$ , where  $b$  is the value returned by LOCAL\_BLKCNT(ARRAY, DIM [, PROC])

### Return Value

The value of LOCAL\_UINDEX(ARRAY, DIM, PROC) has a value whose  $i$ th component is the local index of the last element of the  $i$ th block in dimension DIM of ARRAY on processor PROC.

LOCAL\_UINDEX(ARRAY, DIM) returns the same value as LOCAL\_UINDEX(ARRAY, DIM, PROC=MY\_PROCESSOR()).

## MY\_PROCESSOR

Pure function returns the identifying number of the calling physical processor.

### Synopsis

```
MY_PROCESSOR( )
```

### Return Type

The result is scalar and of type default integer.

### Return Value

Returns the identifying number of the physical processor from which the call is made. This value is in the range where is the value returned by NUMBER\_OF\_PROCESSORS().

## PHYSICAL\_TO\_ABSTRACT

Subroutine returns coordinates for an abstract processor, relative to a global actual argument array, corresponding to a specified physical processor. This procedure can be used only on systems where there is a one-to-one correspondence between abstract processors and physical processors. On systems where this correspondence is one-to-many an equivalent, system-dependent procedure should be provided.

### Synopsis

```
PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)
```

### Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

PROC must be scalar and of type default integer. It is an INTENT(IN) argument. It contains an identifying value for a physical processor.

INDEX must be a rank-1 integer array. It is an INTENT(OUT) argument. The size of INDEX must equal the rank of the processor arrangement onto which the global HPF array is mapped. INDEX receives the coordinates within this processors arrangement of the abstract processor associated with the physical processor specified by PROC.



# Index

## Symbols

-Miomutex, 274  
-mp, 274  
-Mreentrant, 274

## A

ABSTRACT\_TO\_PHYSICAL, 324  
ACCEPT, 41  
ADVANCE, 148  
ALIGN, 316  
    WITH, 316  
ALLOCATABLE, 42  
ALLOCATE, 43  
arithmetic expressions, 9  
ARRAY, 44  
ARRAY attribute, 132  
arrays  
    ARRAY attribute, 132  
    assumed shape, 130  
    assumed size, 130  
    CM Fortran constructors, 133  
    constructor extensions, 133  
    constructors, 132  
    deferred shape, 130  
    explicit shape, 130  
    sections, 131, 132  
    specification, 130  
    specification assumed shape, 130  
    specification assumed size, 131  
    specification deferred shape, 131  
    specification explicit shape, 130  
    subscripts, 131  
    subscript triplets, 131

    vector subscripts, 132  
ASSIGN, 45  
assignment statements, 12  
assumed shape arrays, 130  
assumed size arrays, 130  
ATOMIC directive, 282  
attribute  
    DIMENSION, 317  
    PURE, 106

## B

BACKSPACE, 46  
    specifier  
        ERR, 46  
        IOSTAT, 46  
        UNIT, 46  
BARRIER directive, 282  
Barriers  
    explicit, 271  
    implicit, 271  
binary constants, 25  
BLOCK, 318  
BLOCKDATA, 46  
BYTE, 47

## C

C  
    c\_f\_pointer, 305, 307, 308  
    interoperability, 305  
    pointer types, 305  
c\_f\_pointer, 305, 307, 308  
CALL, 47  
CASE, 49  
character  
    scalar memory reference, 35  
CHARACTER, 49  
character constants, 22  
character set  
    C language compatibility, 3  
Clauses  
    directives, 273  
    driectives, 274  
CLOSE, 50  
    DISPOSE specifier, 51  
    DISP specifier, 51  
    ERR specifier, 51

    IOSTAT specifier, 51  
    STATUS specifier, 51  
    UNIT specifier, 51  
closing a file, 137  
CM Fortran  
    arrays, 132  
CM Fortran Intrinsic, 313  
    CSHIFT, 313  
    EOSHIFT, 313  
    RESHAPE, 314  
column formatting  
    continuation field, 4, 5  
    label field, 4, 5  
    statement field, 4, 5  
COMMON, 51  
COMPLEX, 54  
complex constants, 22, 22  
Conformance to standards, xxiii  
constants, 20  
    PARAMETER statement, 23  
constructors, 133  
CONTAINS, 55  
CONTINUE, 56  
Conventions, xxv  
CRITICAL directive, 283  
CSHIFT  
    CM Fortran, 313  
CYCLE, 56  
CYCLIC, 318

## D

DATA, 56  
data types  
    binary constants, 25  
    character constants, 22  
    complex constants, 22, 22  
    constants, 20  
    double precision constants, 21  
    extensions, 18  
    hexadecimal constants, 25  
    integer constants, 20  
    kind parameter, 17  
    logical constants, 22  
    octal constants, 25  
    real constants, 21  
    size specification, 18

DEALLOCATE, 57  
     STAT specifier, 57  
 debug statements, 5  
 DECODE, 58  
 deferred shape arrays, 130  
 derived types, 23  
 DIMENSION, 59  
 direct access files, 136  
 Directives  
     ATOMIC, 282  
     BARRIER, 282  
     clauses, 273, 274  
     CRITICAL...END CRITICAL, 283  
     Fortran parallization overview, 273  
     -Miomutex option, 274  
     -mp option, 274  
     -Mreentrant option, 274  
     Parallelization, 267  
     parallelization, 273  
     recognition, 274  
     Summary table, 281  
 DO, 60  
 DOACROSS directive, 284  
 DO directive, 284  
 DOUBLECOMPLEX, 62  
 DOUBLEPRECISION, 63  
 double precision constants, 21  
 DOWHILE, 62  
 DYNAMIC, 318  
  
**E**  
 ELSE, 64, 81, 82  
 ELSEIF, 81  
 ELSE IF, 65, 82  
 ELSEWHERE, 124  
 ELSE WHERE, 65, 66  
 ENCODE, 66  
 END, 67  
 ENDBLOCKDATA, 46  
 ENDCASE, 49  
 ENDFORALL, 75  
 ENDFUNCTION, 78  
 ENDIF, 81  
 END IF, 82  
 ENDINTERFACE, 90  
  
 ENDPROGRAM, 105  
 ENDSUBROUTINE, 117  
 ENDTYPE, 119  
 ENDWHERE, 124  
 ENTRY, 71  
 Environment variables  
     MPSTKZ, 302  
     OMP\_STACK\_SIZE, 301, 302  
     OMP\_THREAD\_LIMIT, 302  
     OMP\_WAIT\_POLICY, 301, 302  
     OpenMP, 300  
     OpenMP, OMP\_DYNAMIC, 301  
     OpenMP,  
     OMP\_MAX\_ACTIVE\_LEVELS, 301  
     OpenMP, OMP\_NESTED, 301  
     OpenMP, OMP\_NUM\_THREADS,  
     301  
     OpenMP, OMP\_SCHEDULE, 302  
     OpenMP, OMP\_STACK\_SIZE, 302  
     OpenMP, OMP\_THREAD\_LIMIT,  
     302  
     OpenMP, OMP\_WAIT\_POLICY,  
     302  
 EOSHIFT  
     CM Fortran, 313  
 EQUIVALENCE, 73  
 Examples  
     OpenMP Task in Fortran, 272  
 EXIT, 74  
 expressions, 8  
 EXTERNAL, 74  
 EXTRINSIC, 74  
  
**F**  
 F2003 Functions  
     GET\_COMMAND\_ARGUMENT,  
     175, 184, 185, 186  
     IS\_IOSTAT\_END, 193  
     IS\_IOSTAT\_EOR, 193  
     MOVE\_ALLOC, 207  
     NEW\_LINE, 209  
 F2003 Intrinsic  
     EXTENDS\_TYPE\_OF, 183  
     SAME\_TYPE\_AS, 218  
 F77 3F Routines, 239  
     ABORT, 240, 240  
  
 ACCESS, 240  
 ALARM, 241  
 BESSEL FUNCTIONS, 241  
 chdir, 242  
 CHMOD, 242  
 CTIME, 242  
 DATE, 242  
 DRANDM, 254  
 DSECNDS, 256  
 ELAPSED TIME, 243  
 ERROR FUNCTIONS, 243  
 EXIT, 243  
 FDATE, 244  
 FGETC, 244  
 FLUSH, 244  
 FORK, 244  
 FSTAT, 259  
 FSTAT64, 259  
 GERROR, 246  
 GETARG, 246  
 GETC, 247  
 GETCWD, 247  
 GETENV, 247  
 GETGID, 248  
 GETLOG, 248  
 GETPID, 248  
 GETUID, 248  
 GMTIME, 248  
 HOSTNM, 249  
 IARG, 246  
 IDATE, 249  
 IERRNO, 249  
 IOINIT, 249  
 IRAND, 254  
 IRANDM, 254  
 ISATTY, 250  
 ITIME, 250  
 KILL, 250  
 LINK, 251  
 LNBLNK, 251  
 LOC, 251  
 LSTAT, 259  
 LTIME, 251  
 MALLOC, 252  
 MCLOCK, 252  
 MVBITS, 252



- OUTSTR, 252
- PERROR, 253
- PUTC, 253
- PUTENV, 253
- QSORT, 253
- RAND, 254
- RANDOM, 254
- RANGE, 255
- RENAME, 255
- RINDEX, 256
- SECNDS, 256
- SETVBUF, 256
- SETVBUF3F, 257
- SIGNAL, 258
- SLEEP, 258
- SRAND, 254
- STAT, 259
- STIME, 259
- SYMLNK, 259
- SYSTEM, 260
- TIME, 260
- TIMES, 260
- TTYNAM, 261
- UNLINK, 261
- WAIT, 261
- F77 Functions
  - IAND, 188
  - IBCLR, 188
  - IBITS, 188
- F77 VAX/VMS Subroutines, 262
  - DATE, 262
  - EXIT, 262
  - GETARG, 263
  - IARGC, 263
  - IDATE, 263
  - MVBITS, 263
  - RAN, 264
  - SECNDS, 264
  - TIME, 264
- F77 VAX Built-In Functions, 262
  - %LOC, 262
  - %REF(a), 262
- F77 VAX System Subroutines, 261
- F90 Functions
  - ABS, 165
  - ACHAR, 166
  - ACOS, 166, 166
  - ADJUSTL, 167
  - ADJUSTR, 167
  - AIMAG, 167
  - AINT, 168
  - ALL, 168
  - ALLOCATED, 169
  - ANINT, 169
  - ANY, 170
  - ASIN, 170, 170
  - ASSOCIATED, 171
  - ATAN, 171, 172
  - ATAN2, 172, 172
  - BIT\_SIZE, 173
  - BTEST, 173
  - CEILING, 173
  - CHAR, 174
  - CMPLX, 174, 179
  - CONJG, 175
  - COSH, 176, 176, 177
  - COUNT, 177
  - CSHIFT, 178
  - DATE\_AND\_TIME, 178
  - DBLE, 179
  - DIGITS, 180
  - DIM, 180
  - DOT\_PRODUCT, 180
  - DPROD, 181
  - EOSHIFT, 181
  - EPSILON, 182
  - EXP, 182
  - EXPONENT, 183
  - FLOOR, 184, 189, 189, 189
  - FRACTION, 184
  - HUGE, 187, 226
  - IACHAR, 187
  - IAND, 190, 192, 199, 199, 199, 199, 200
  - IBITS, 169, 175, 182, 209, 211, 229
  - INDEX, 190
  - INT, 190, 191, 192, 195
  - IOR, 192
  - ISHFT, 192, 201, 217, 221
  - ISHFTC, 194
  - KIND, 196
  - LBOUND, 196
  - LEN, 198
  - LEN\_TRIM, 198
  - LGE, 198
  - LIT, 200
  - LOG, 200, 200
  - LOG10, 201
  - LOGICAL, 201
  - MATMUL, 202
  - MAX, 202
  - MAXEXPONENT, 203
  - MAXLOC, 203
  - MAXVAL, 204, 206
  - MERGE, 204
  - MIN, 205
  - MINEXPONENT, 205
  - MINVAL, 206
  - MOD, 207
  - MODULO, 207
  - MVBITS, 208
  - NEAREST, 208
  - NINT, 191, 195, 196, 210
  - NOT, 210
  - PACK, 211, 228
  - POPCNT, 211, 212
  - PRECISION, 213
  - PRESENT, 213
  - PRODUCT, 214
  - RADIX, 214
  - RANDOM\_NUMBER, 214
  - RANDOM\_SEED, 215
  - RANGE, 215
  - REAL, 216
  - REPEAT, 216
  - RESHAPE, 217
  - RRSPACING, 217
  - SCALE, 218
  - SCAN, 218
  - SELECTED\_INT\_KIND, 219
  - SELECTED\_REAL\_KIND, 219
  - SET\_EXPONENT, 220
  - SHAPE, 220
  - SIGN, 221
  - SIN, 222, 222
  - SINH, 222
  - SIZE, 223

SPACING, 223  
 SPREAD, 223  
 SQRT, 224  
 SUM, 224  
 SYSTEM\_CLOCK, 225  
 TAN, 225, 226  
 TANH, 226  
 TRANSPOSE, 227  
 TRIM, 227  
 UBOUND, 227  
 UNPACK, 228  
 VERIFY, 195, 228, 229  
**F95 Functions**  
   CPU\_TIME, 177  
   NULL, 210  
 file access methods, 135  
 fixed source form, 1, 4  
 FLUSH directive, 286  
 FORALL, 75  
 FORMAT, 76  
 Format control  
   specifier  
     \$ specifier, 148  
     A specifier, 141  
     BN specifier, 145  
     B specifier, 142  
     D specifier, 142  
     end of record, 148  
     EN specifier, 143  
     E specifier, 143  
     ES specifier, 143  
     format termination, 148  
     F specifier, 143  
     G specifier, 144  
     H specifier, 145  
     I specifier, 144  
     L specifier, 144  
     O specifier, 145, 147  
     P specifier, 146  
     Q specifier, 146  
     quote control, 144  
     slash, 148  
     SP specifier, 146  
     S specifier, 146  
     SS specifier, 146  
     TL specifier, 147  
     T specifier, 147  
     X specifier, 147  
     Z specifier, 145, 147  
 format specifications, 140  
 formatted data transfer, 139  
 Fortran 77  
   Math Intrinsic, 153  
 Fortran Intrinsic, 153  
 Fortran Parallelization Directives  
   DOACROSS, 284, 284  
   ORDERED, 287  
 Fortran program unit  
   elements of, 1  
 free source form, 1, 4  
   comments, 4  
   continuation line, 4  
   statement labels, 4, 4  
 FUNCTION, 78  
  
**G**  
 GLOBAL\_ALIGNMENT, 324  
 GLOBAL\_DISTRIBUTION, 324  
 GLOBAL\_LBOUND, 324  
 GLOBAL\_SHAPE, 325  
 GLOBAL\_SIZE, 325  
 GLOBAL\_TEMPLATE, 326  
 GLOBAL\_TO\_LOCAL, 326  
 GLOBAL\_UBOUND, 327  
 GOTO  
   Assigned, 79  
   Computed, 80  
   Unconditional, 80  
  
**H**  
 hexadecimal constants, 25, 26  
 hollerith constants, 27  
 HPF\_LOCAL Functions  
   ABSTRACT\_TO\_PHYSICAL, 324  
   GLOBAL\_ALIGNMENT, 324  
   GLOBAL\_DISTRIBUTION, 324  
   GLOBAL\_LBOUND, 324  
   GLOBAL\_SHAPE, 325  
   GLOBAL\_SIZE, 325  
   GLOBAL\_TEMPLATE, 326  
   GLOBAL\_TO\_LOCAL, 326  
   GLOBAL\_UBOUND, 327  
   LOCAL\_BLKCNT, 328  
   LOCAL\_LINDEX, 328  
   LOCAL\_TO\_GLOBAL, 329  
   LOCAL\_UNINDEX, 329  
   MY\_PROCESSOR, 330  
   Overview, 322  
   PHYSICAL\_TO\_ABSTRACT, 330  
 HPF Directives  
   !HPF\$, 314  
   \*HPF\$, 314  
   adding to HPF, 314  
   ALIGN, 316  
   CHPF\$, 314  
   DISTRIBUTE, 318  
   DISTRIBUTE BLOCK, 318  
   DISTRIBUTE CYCLIC, 318  
   DISTRIBUTE ONTO, 318  
   INDEPENDENT, 319  
   INHERIT, 320  
   NOSEQUENCE, 321  
   PROCESSORS, 320  
   REALIGN, 316  
   REDISTRIBUTE, 318  
   SEQUENCE, 321  
   summary table, 315  
   TEMPLATE, 322  
  
**I**  
 IF  
   Arithmetic, 81  
   Block, 81  
   Logical, 82  
 IMPLICIT, 82  
 implied DO list, 139  
 IMPORT, 83  
 INCLUDE, 6, 84  
 INDEPENDENT, 319  
 INHERIT, 320  
 input and output, 135  
 INQUIRE, 85  
   ACCESS specifier, 85  
   ACTION specifier, 85  
   BLANK specifier, 85  
   DELIM specifier, 85  
   DIRECT specifier, 85  
   ERR specifier, 85

- EXIST specifier, 85
- FILE specifier, 85
- FORMATTED specifier, 86
- FORM specifier, 86
- IOSTAT specifier, 86
- NAMED specifier, 86
- NAME specifier, 86
- NEXTREC specifier, 86
- NUMBER specifier, 86
- OPENED specifier, 86
- PAD specifier, 86
- PENDING specifier, 86
- POSITION specifier, 87
- POS specifier, 87, 108, 126
- READ specifier, 87
- READWRITE specifier, 87
- RECL specifier, 87
- SEQUENTIAL specifier, 87
- STATUS specifier, 88
- UNFORMATTED specifier, 88
- WRITE specifier, 88
- integer
  - scalar memory reference, 35
- INTEGER, 88
- integer constants, 20
- INTENT, 89
- INTERFACE, 90
- Interoperability
  - with C, 305
- Interoperability with C, 305, 307, 308
- Intrinsics
  - Tables, 153
- INTRINSIC, 91
- intrinsic data types, 17
- L**
- LEADZ, 197
- LEN, 198
- Libraries
  - run-time routines, 296
- list-directed formatting, 149
- list-directed input, 149
- list-directed output, 150
- LOCAL\_BLKCNT, 328
- LOCAL\_LINDEX, 328
- LOCAL\_TO\_GLOBAL, 329
- LOCAL\_UNIDEX, 329
- logical
  - scalar memory reference, 35
- LOGICAL, 92
- logical constants, 22
- M**
- MAP@, 93
- MASTER directive, 286
- multiple statements, 4, 4
- MY\_PROCESSOR, 330
- N**
- NAMELIST, 95
- namelist groups, 151
- namelist input, 151
- namelist output, 152
- non-advancing i/o, 148
- NULLIFY, 95
- O**
- obsolescent, 35
- octal constants, 25, 26
- OMP\_DYNAMIC, 300, 301
- omp\_get\_ancestor\_thread\_num(), 296
- OMP\_MAX\_ACTIVE\_LEVELS, 300, 301
- OMP\_NESTED, 301, 301
- OMP\_NUM\_THREADS, 301, 301
- OMP\_SCHEDULE, 301, 302
- OMP\_STACK\_SIZE, 301, 302
- OMP\_THREAD\_LIMIT, 301, 302
- OMP\_WAIT\_POLICY, 302
- ONTO, 318
- OPEN, 96
  - ACCESS specifier, 96
  - ACTION specifier, 96
  - ASYNCHRONOUS specifier, 96
  - BLANK specifier, 96
  - DELIM specifier, 97
  - ERR specifier, 97
  - FILE specifier, 97
  - FORM specifier, 97
  - IOSTAT specifier, 97
  - PAD specifier, 97
  - POSITION specifier, 97
  - RECL specifier, 97
  - STATUS specifier, 97
- opening and closing files, 136
- OpenMP
  - barrier, 271
  - environment variables, 300
  - task, 270, 271
  - task scheduling, 271
  - taskwait, 271
- OpenMP environment variables
  - MPSTKZ, 302
  - OMP\_DYNAMIC, 300, 301
  - OMP\_MAX\_ACTIVE\_LEVELS, 300, 301
  - OMP\_NESTED, 301, 301
  - OMP\_NUM\_THREADS, 301, 301
  - OMP\_SCHEDULE, 301, 302
  - OMP\_THREAD\_LIMIT, 301
- OpenMP Fortran Directives, 267
  - ATOMIC, 282
  - BARRIER, 282
  - CRITICAL, 283
  - DO, 284
  - FLUSH, 286
  - MASTER, 286
  - ORDERED, 287
  - PARALLEL, 288
  - PARALLEL DO, 289, 289
  - PARALLEL SECTIONS, 290
  - PARALLEL WORKSHARE, 290, 291
  - SECTIONS, 291
  - SINGLE, 292
  - TASK, 292, 294
  - THREADPRIVATE, 294
  - WORKSHARE, 295
- OpenMP Fortran Support Routines
  - omp\_destroy\_lock(), 299
  - omp\_get\_ancestor\_thread\_num(), 296
  - omp\_get\_dynamic(), 298
  - omp\_get\_level(), 297, 297
  - omp\_get\_max\_threads(), 297
  - omp\_get\_nested(), 298
  - omp\_get\_num\_procs(), 297

omp\_get\_num\_threads(), 296  
 omp\_get\_schedule(), 299, 299  
 omp\_get\_stack\_size(), 297  
 omp\_get\_team\_size(), 298  
 omp\_get\_thread\_num(), 296  
 omp\_get\_wtick(), 299  
 omp\_get\_wtime(), 299  
 omp\_in\_parallel(), 298  
 omp\_init\_lock(), 299  
 omp\_set\_dynamic(), 298  
 omp\_set\_lock(), 300  
 omp\_set\_nested(), 298  
 omp\_set\_num\_threads(), 296  
 omp\_set\_stack\_size(), 297  
 omp\_test\_lock(), 300  
 omp\_unset\_lock(), 300  
 option  
   -Mdlines, 5  
   -Mfreeform, 1  
 OPTIONAL, 100  
 OPTIONS, 100  
 ORDERED directive, 287

## P

PARALLEL directive, 288  
 PARALLEL DO directive, 289  
 Parallelization  
   Directives, defined, 273  
   directives format, 273  
 Parallelization Directives, 267  
 PARALLEL SECTIONS directive, 290  
 PARALLEL WORKSHARE directive,  
 290  
 PARAMETER, 101  
 PAUSE, 102  
 PHYSICAL\_TO\_ABSTRACT, 330  
 POINTER, 102, 102  
 pointers, 25  
 Pointers  
   interoperability with C, 305  
 precedence rules, 8  
 PRINT, 103  
 PRIVATE, 104  
 PROGRAM, 105  
 PUBLIC, 106  
 PURE, 106

## R

READ, 107  
   ADVANCE specifier, 107  
   ASYNCHRONOUS specifier, 107,  
   125  
   BLANK specifier, 107  
   END specifier, 107  
   EOR specifier, 107  
   ERR specifier, 107  
   FMT specifier, 107  
   IOSTAT specifier, 107  
   NML specifier, 107  
   PAD specifier, 108  
   REC specifier, 108  
   SIZE specifier, 108  
 REAL, 109  
 real constants, 21  
 REALIGN, 316  
 RECORD, 110  
 RECURSIVE, 111  
 REDIMENSION, 111  
 REDISTRIBUTE, 318  
 Related Publications, xxvi  
 RESHAPE  
   CM Fortran, 314  
 RETURN, 112  
 REWIND, 113  
   specifier  
     ERR, 113  
     IOSTAT, 113  
     UNIT, 113  
 Run-time  
   library routines, 296

## S

scalar memory reference  
   character, 35  
   integer, 35  
   logical, 35  
 SECTIONS directive, 291  
 SELECT, 49  
 SELECT CASE, 114  
 SINGLE directive, 292  
 Standard compatibility, xxiii  
 standard preconnected units, 136  
 statement

obsolescent, 35  
 origin, 35

## Statement

ACCEPT, 41  
 ALLOCATABLE, 42  
 ALLOCATE, 43  
 ARRAY, 44, 132  
 ASSIGN, 45  
 BACKSPACE, 46  
 BLOCKDATA, 46  
 BYTE, 47  
 CALL, 47  
 CASE, 49  
 CHARACTER, 49  
 CLOSE, 50  
 COMMON, 51  
 COMPLEX, 54  
 CONTAINS, 55  
 CONTINUE, 56  
 CYCLE, 56  
 DATA, 56  
 DEALLOCATE, 57  
 DECODE, 58  
 DIMENSION, 59  
 DO, 60  
 DOUBLECOMPLEX, 62  
 DOUBLEPRECISION, 63  
 DOWHILE, 62  
 ELSE, 64, 81, 82  
 ELSE IF, 65, 81, 82  
 ELSEWHERE, 124  
 ELSE WHERE, 65, 66  
 ENCODE, 66  
 END, 67  
 ENDBLOCKDATA, 46  
 ENDCASE, 49  
 ENDFORALL, 75  
 END FUNCTION, 78  
 END IF, 81, 82  
 ENDINTERFACE, 90  
 END PROGRAM, 105  
 ENDSUBROUTINE, 117  
 ENDTYPE, 119  
 ENDWHERE, 124  
 ENTRY, 71  
 EQUIVALENCE, 73

- EXIT, 74
  - EXTERNAL, 74
  - EXTRINSIC, 74
  - FORALL, 75
  - FORMAT, 76
  - FUNCTION, 78
  - GOTO, 79, 80, 80
  - IF, 81, 81, 82
  - IMPLICIT, 82
  - IMPORT, 83
  - INCLUDE, 84
  - INQUIRE, 85
  - INTEGER, 88
  - INTENT, 89
  - INTERFACE, 90
  - INTRINSIC, 91
  - LOGICAL, 92
  - MAP@, 93
  - NAMELIST, 95
  - NULLIFY, 95
  - OPEN, 96
  - OPTIONAL, 100
  - OPTIONS, 100
  - PARAMETER, 101
  - PAUSE, 102
  - POINTER, 102, 102
  - PRINT, 103
  - PRIVATE, 104
  - PUBLIC, 106
  - READ, 107
  - REAL, 109
  - RECORD, 110
  - RECURSIVE, 111
  - REDIMENSION, 111
  - RETURN, 112
  - REWIND, 113
  - SELECT, 49
  - SELECT CASE, 114
  - SEQUENCE, 115
  - STOP, 115
  - STRUCTURE@, 116
  - SUBROUTINE, 117
  - TARGET, 118
  - THEN, 81, 82, 118
  - TYPE, 119
  - UNION@, 119
  - USE, 121
  - VOLATILE, 122
  - WAIT, 123
  - WHERE, 124
  - WRITE, 125
  - Statement ordering, 2
  - Statements and comments, 1
  - STOP, 115
  - STRUCTURE@, 116
  - SUBROUTINE, 117
  - symbolic name scope, 12
- T**
- tab formatting, 6
  - Tables
    - Intrinsics, 153
  - TARGET, 118
  - targets, 25
  - TASK directive, 292, 294
  - Tasks
    - construct, 272
    - Fortran example, 272
    - OpenMP overview, 270, 271
    - scheduling points, 271
    - taskwait call, 271
  - THEN, 81
  - THREADPRIVATE directive, 294
  - TYPE, 119
- U**
- unformatted data transfer, 139, 139
  - UNION@, 119
  - USE, 121
- V**
- VOLATILE, 122
- W**
- WAIT, 123
  - WHERE, 124
  - WITH, 316
  - WORKSHARE directive, 295
  - WRITE
    - specifier
      - ADVANCE specifier, 125
      - DELIM specifier, 125
- ERR specifier, 126
  - FMT specifier, 126
  - IOSTAT specifier, 126
  - NML specifier, 126
  - REC specifier, 126

